

Lego Boost Roboter steuern mit Python unter Windows oder Linux

Inhaltsverzeichnis

Verwendete Betriebssysteme und Softwareversionen:	1
0 Einleitung	2
1 Roboterhardware des Lego Boost	3
2 Bluetoothkommunikation	3
2.1 Testen der Bluetoothkommunikation unter Linux mit gatttool	4
2.1.1 Auslesen der vom Move Hub angebotenen GATT „Primary Services“	4
2.1.2 Auslesen der vom Move Hub angebotenen GATT „Characteristics“	5
2.1.3 Auslesen der vom Move Hub angebotenen Descriptors der Characteristics	5
2.1.4 Konkrete Beispiel der BLE-Kommunikation mit gatttool	6
2.2 Testen der Bluetoothkommunikation unter Linux mit Python und der Bibliothek bluepy	7
2.2.1 Auslesen der vom Move Hub angebotenen GATT „Primary Services“	8
2.2.2 Auslesen der vom Move Hub angebotenen GATT „Characteristics“	8
2.2.3 Auslesen der vom Move Hub angebotenen Descriptors der Characteristics	8
2.2.4 Konkrete Beispiel der BLE-Kommunikation mit bluepy	9
3 Python-Bibliotheken für die Bluetooth Low Energy Kommunikation mit dem Protokoll GATT	11
3.1 BLE Python-Bibliothek gatt (nur Linux)	12
3.2 BLE Python-Bibliothek gattlib (nur Linux)	12
3.3 BLE Python-Bibliothek pygatt (Linux und Windows)	13
3.4 BLE Python-Bibliothek bluepy (nur Linux)	13
4 Bibliothek pylegoboost (Verwendung von pygatt für Windows/Linux und gatt für Windows)	14
5 Anhang	15
5.1 Klassendiagramm pylegoboost Bibliothek	15
5.2 Programmablaufplan Instanziierung MoveHub-Objekt	16
5.3 Programmablaufplan Sensordaten empfangen	17
5.4 Fachbücher	18

Verwendete Betriebssysteme und Softwareversionen:

PC mit Windows 7, 64 Bit und Python 3.63, Raspberry Pi 3 mit Raspbian GNU/Linux 9 (stretch) und Python 3.5.3., mit Raspberry Pi Zero derzeit noch Probleme mit BLE-Kommunikation (Raspbian stretch lite Version)

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 auch **teils kompatibel mit dem WeDo** Robotiksystem von Lego Education®, welches aber ausschließlich für den Schulbereich gedacht ist.

Das Konzept vom Lego Boost sieht vor, dass dieser Roboter **ausschließlich zusammen mit einem Tablet genutzt** wird, welches als Master-Rechner für die Steuerung, für die Programmierung als auch für die Darstellung der Bauanleitungen verwendet wird¹. Hier 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 ist leider konzeptionell nicht vorgesehen.**

Andere Programmierumgebungen als diese grafikbasierte und sehr rudimentäre auf dem Tablet werden von Lego bewusst nicht angeboten und auch nicht unterstützt. Dies ist vermutlich mit ein Grund für den mangelnden Erfolg dieses Robotiksets, zumal die Lego-Programmierungsumgebung auch neben iPads nur auf sehr wenigen Android-Tablets verwendet werden kann. Dies führte letztlich dazu, dass das Lego Boost Set derzeit für ca. 110 € statt UVP 159,90 € von den meisten Onlinehändlern angeboten wird.

Zum Glück gibt es aber kreative Hacker, die die unverschlüsselte Bluetooth Low Energy (BLE) Kommunikation der Lego-App mit dem Boost Roboters belauscht haben. Mit dem dadurch gewonnen Wissen wurden von mehreren Personen quelloffene Softwarebibliotheken erstellt. Dadurch ist es jetzt möglich, den **Lego Boost 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.

Das Programm wird hierbei 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 ab, der beispielsweise ein Windows-PC, ein Apple-PC, ein Raspberry Pi oder ein virtueller PC in der Cloud sein kann. Der Computer muss nur einen sogenannten Python-Interpreter besitzen.

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 gibt. Außerdem **ist Python leicht zu erlernen** und bietet optimale Voraussetzungen, um das objektorientierte Programmieren zu lernen.

Achtung:

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.

Achtung, Achtung:

Viele Programmierbeispiele im Internet gerade im Zusammenhang mit BLE oder Lego Boost beziehen sich auf Python 2.7. Neben einer unterschiedlichen Syntax für *print*-Befehle, unterscheiden sich beide Pythonversionen gerade bei der Kommunikation mit Bytefolgen, wie sie in BLE nötig ist²: In Python 2 werden für Bytefolgen ASCII-Strings verwendet. Python 3 verwendet jedoch den hier neuen Typ „Byte Array“³.

1 Neuerdings soll es jedoch die Tablet-App auch für Windows 10 Rechner geben.

2 Siehe z.B.: sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

3 Z.B. wird die Bytefolge 00 10 in Python 3 mit dem Byte Array *b'\x00\x10'* bezeichnet.

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). Der Computer, mit dem der Move Hub via BLE gesteuert werden soll, muss also eine BLE-Schnittstelle besitzen. Ein Raspberry Pi 3 besitzt diese Schnittstelle auf dessen Platine fest verbaut und im aktuellen Betriebssystem Raspian unterstützt (Linux BlueZ Service).

Verfügt der Steuercomputer über keine interne BLE-Schnittstelle, so hilft der BLE-Dongle der Fa. BlueGiga (Modell BLED112): Dieser Dongle wird in einen USB-Anschluss des Steuercomputers eingesteckt und 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 BlueGiga Dongle BLED112 kostet ca. 12 €⁴. Andere BLE-Dongles funktionieren nur bei bestimmten Betriebssystemversionen, im Fall von Windows z.B. meistens erst ab Windows 8.

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 inkl. Kopplung (Pairing) 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 ⁵.

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 angeregt.

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/Cleint muss dann selbst in Erinnerung behalten, was der letzte gesendete Status des Tasters ist. **Diese Art der eventbasierten Kommunikation/Programmierung erspart viel Kommunikationsarbeit**, da ja dieser 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.

⁴ Er ist in Deutschland nicht leicht zu bekommen. Daher am besten aus China via Aliexpress bestellen.

⁵ golem.de/news/golem-de-programmiert-bluetoothle-im-eigenbau-1404-105896.html

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 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 von dem Portugiesen Jorge Pereira und Russen Andrey Pokhilko „gehackt“**. Beide erstellten aus diesem Informationen die Pythonbibliotheken *pyb00st*⁶ bzw. *pylgbst*⁷.

Mit den aus diesen Bibliotheken vorhandenem Wissen und dem Linux Tool *gatttool* kann man die BLE-Kommunikation mit dem Move Hub sehr gut von der Linux Kommandozeile testen.

2.1 Testen der Bluetoothkommunikation unter Linux mit *gatttool*⁸

Mit dem Befehl *hciconfig* 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: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x0005, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001623-1212-efde-1623-785feabcd123
```

Die innerhalb der BLE-Spezifikation vordefinierten gelb unterlegten Services (16-Bit UUID) sind: „1800“ steht für „Generic Access“ und „1801“ für „Generic Attribute“. Diese beiden Services muss jedes BLE-Gerät anbieten.

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

6 github.com/undera/pylgbst

7 github.com/JorgePe/pyb00st

8 Es gibt übrigens auch eine sehr interessante APP für Android, die ähnliche Funktionen wie *gatttool* aufweist: „nRF Connect“ von Nordic Semiconductor

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: 00002a05-0000-1000-8000-00805f9b34fb
... gehört zum Service 1800

handle: 0x0006, char properties: 0x4e, char value handle: 0x0007, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0008, char properties: 0x4e, char value handle: 0x0009, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x000a, char properties: 0x02, char value handle: 0x000b, uuid: 00002a04-0000-1000-8000-00805f9b34fb
... gehören zum Service 1801

handle: 0x000d, char properties: 0x1e, char value handle: 0x000e, uuid: 00001624-1212-efde-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 Attribute kann über ein 8-Bit „Handle“ zugegriffen 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.

0x1e ist gleich b11110: Die erste 1 steht für „Notify“ 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“ Option⁹.

Beim Move Hub läuft abgesehen von der Device ID die gesamte Kommunikation über den handle 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 ¹⁰ oder ¹¹.

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

⁹ bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.attribute.gatt.characteristic_declaration.xml

¹⁰ Robert Davidson et al.: Getting Started with Bluetooth Low Energy. O'Reilly, Sebastopol, 2015. [safaribookonline.com/library/view/getting-started-with/9781491900550/ch04.html](https://safaribooksonline.com/library/view/getting-started-with/9781491900550/ch04.html)

¹¹ Naresh Gupta: Inside Bluetooth Low Energy. Artech House, Boston, 2013.

...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-Spezifikation¹² 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/Indications¹³. Mit 0x0100 aktiviert man z.B. Notifications bzw. mit 0x1000 entsprechend Indications.

Die UUID „00001624-...“ ist eine herstellerspezifische UUID. Über den Characteristic Value Handle 0x0e des läßt 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.

Bei den folgenden drei Beispielen 1)-3) wird jeweils die Characteristic mit dem Handle 0x0e mit einer Bytefolge (in Python „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.

2.1.4 Konkrete Beispiel der BLE-Kommunikation mit gatttool

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
```

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
```

¹² bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.device_name.xml

¹³ bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml

```

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*¹⁴ von Jorge Pereira 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 eignete sich *bluepy*¹⁵ 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 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>
  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: <7500420401800078dbbc096c3d7abdbc096c3c01000000000000>

```

¹⁴ github.com/JorgePe/BOOSTreveng

¹⁵ github.com/getsenic/gatt-python

Nun kommt die Pythonbibliothek *bluepy* zum Einsatz für die Kommunikation mit dem Move Hub.

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

Die nachfolgenden Befehle werden jeweils in die Pythonshell oder in einem separaten 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 Hubs 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 (Handle *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/Indications 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 Parameters>
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-1623-785feabcd123>
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 0xc und 0xd 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 *btile.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 : 0f00043a0128000000001000000002
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 : 08004502ffffff
Notification erhalten : 08004502ffffff
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()) # Instanziierung 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 Tscheuss!')

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 Tscheuss!')
finally:
    dev.disconnect()
```

3 Python-Bibliotheken für die Bluetooth 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 `sudo 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)

Ein Fork von *gattlib* wird auch *pygattlib* genannt. *gattlib* basiert auf einem C-Quellcode der unter GitHub unter *gattlib*¹⁶ 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-dev
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 Pathonbibliothek *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.

```
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),
```

¹⁶ github.com/labapart/gattlib

```

'utf-8').hex()))

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

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

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 Angabe der konkreten Exception. Auch die Verwendung von GATTResponse für das asynchrone Lesen führt zum selben Laufzeitfehler.

Daher kann nach jetzigem Wissen gattlib nicht auf einem Raspberry Pi zur Steuerung des Move Hubs verwendet werden.

3.3 BLE Python-Bibliothek pygatt (Linux und Windows)

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*.

pygatt kann aber auch für Python unter Windows verwendet werden, dann benötigt man BLE-Dongle der Fa. BlueGiga (Modell BLED112). 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 BlueGiga Dongle laufen auf jeder Plattform, die den Dongle einbinden kann, d.h. Treiber für ihn besitzt.

Die Bibliothek *pygatt* wird mittels `sudo pip3 install pygatt` installiert.

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

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

3.4 BLE Python-Bibliothek bluepy (nur Linux)

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 `sudo pip3 install bluepy` installiert.

Mehr Quellcodebeispiele für die Bibliothek *bluepy* ist in Abschnitt Fehler: Verweis nicht gefunden 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 allzu oft zu Abstürzen.

Daher wurde schlussendlich für den Raspberry Pi ohne BlueGiga-Dongle 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 Pokhilko¹⁷ entnommen.

Für die BLE-Kommunikation wurde der BlueGiga-Teil der Bibliothek *pygatt*¹⁸ sowie die Linux-BLE-Bibliothek *gatt*¹⁹ verwendet.

Dadurch ist es möglich, das selbe Pythonprogramm

- **auf einem Windows PC mit BlueGiga Dongle**
- **auf einem Raspberry Pi mit BlueGiga Dongel**
- **und auf einem Raspberry Pi ohne BlueGiga Dongel**

auszuführen.

D.h. hiermit 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 mit fährt.

In der Zip-Datei *pylegoboost_vx* befinden sich Beispielprogramme sowie der Ordner *pylego-boost*, 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. Ansonsten muss man *Python3* den Pfad zur Bibliothek *pylego-boost* mitteilen, da diese ja nicht via *pip3* installiert wurde.

In den Modulen von *pylegoboost* befinden sich noch viele *sleep()*-Befehle. Ohne sie würden bei Linux ohne BlueGiga-Dongle 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.

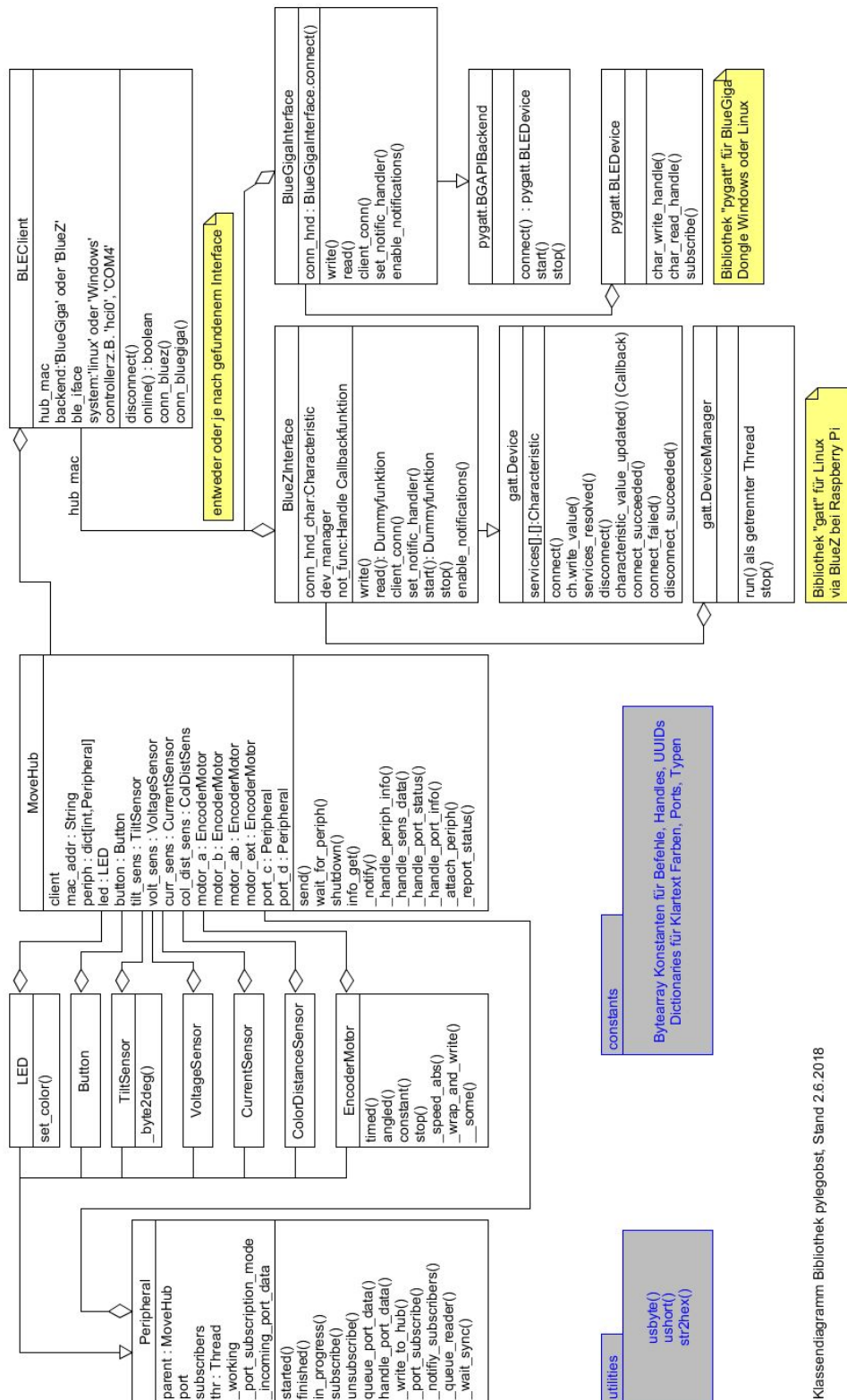
Ein Klassendiagramm von *pylegoboost* ist in Abschnitt 5.1 dargestellt.

¹⁷ Siehe github.com/undera/pylgbst

¹⁸ pypi.org/project/pygatt/3.2.0/

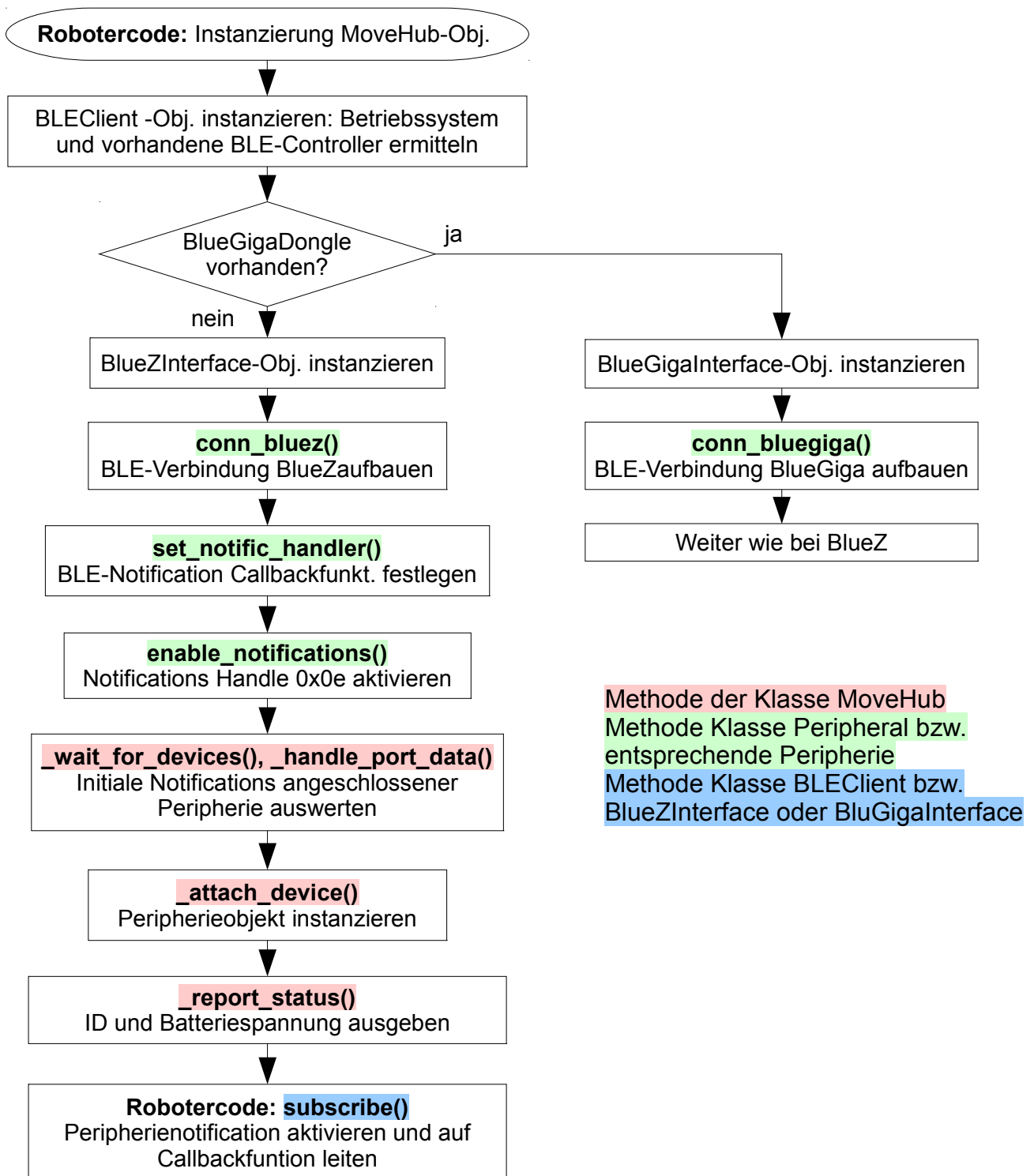
¹⁹ pypi.org/project/gatt/0.2.7/

5.1 Klassendiagramm pylegoboost Bibliothek

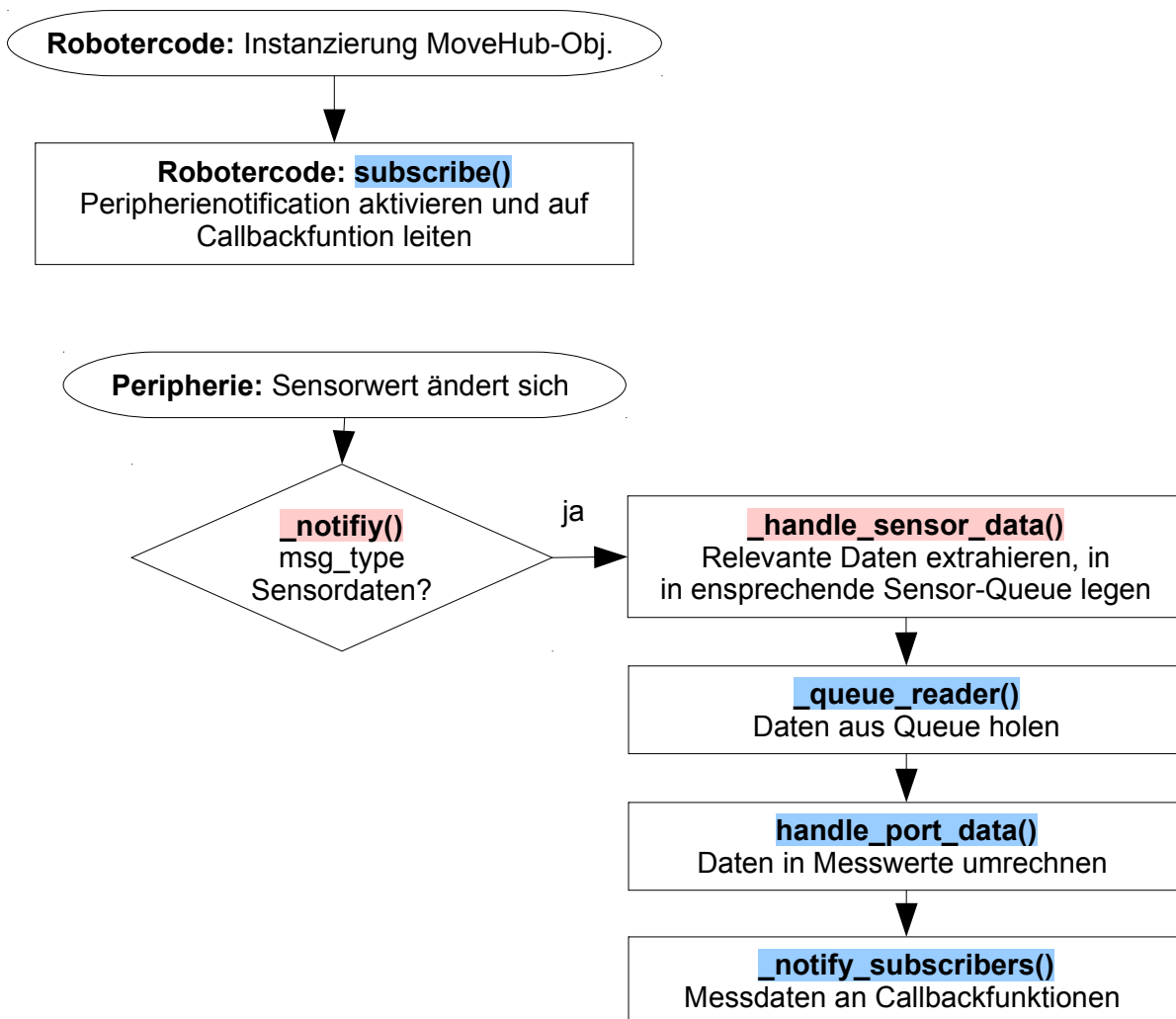


Klassendiagramm Bibliothek pylegobst, Stand 2.6.2018

5.2 Programmablaufplan Instanziierung 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 Fachbücher

Buch	Bemerkung
B. Klein: Einführung in Python 3.	Sehr umfassend, 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 <i>Numpy</i> und <i>Matplotlib</i> .
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. <i>tkinter</i> .
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.