Internet of Things

Report by Chiara Simoni and
Giada Martina Stivala

Some project on debugging Bluetooth Low Energy devices

A.A. 2017/2018
Prof. Luciano Bononi, Marco Di Felice

July 22, 2018

Contents

Chapter 1

Introduction

Bluetooth Low Energy devices are flooding the market of cheap, wearable, health, home-appliance electronics. Some examples include monitoring devices for sport, health and outdoor activities in general; they also connect many household appliances as well as security cameras and locks. Other interesting use-cases include wristbands used in theme parks or live music concerts (although the technology may differ).

Intrinsically, these devices perform very simple tasks, mostly in the field of data gathering. It seems totally harmless to have a smart watch tracking your position at any time of day, but what if it were possible for anyone to query it and extract its data? Their minimalism is what makes them so appealing to the general public, but it is sometimes their intrinsic weakness.

Moving a bit further from *"hackers"* and information security, data gathering is a fundamental step for data analysis in a Big Data context. Privacy concerns should be taken into consideration also in terms of data anonymization.

In this project report, we will start by describing general device characteristics and Bluetooth specifications. We will then move to their concrete observation with hands-on analyses, taking into consideration different developing tools and devices. In particular, we will focus our attention on security risks and concrete flaws, highlighting security exploits when possible. Finally, we will summarize our findings, including a few ideas for future directions and developments.

For our tests, we used a Mi Band 2 smart band, a Magic Blue smart lightbulb and a ST Microelectronics IoT node (model `B-L475E-IOT1A2 Discovery kit`). The code was run both on Ubuntu 16.04 and Kali Linux 4.13. It is worth mentioning that we also tried to use a Ubuntu GNOME OS, virtualized via VMWare on Windows 10, but this option was soon discarded as VMWare does not provide the drivers for Bluetooth Low Energy.

The software at our disposal included the builtin Linux commands, as well as Wireshark and Android apps (namely BLEScanner and nRFConnect). As we will see, we also tested some open-source tools available from Github: bleah, btlejuice and gattacker.

Chapter 2

Bluetooth Low Energy Specification

In this chapter we briefly report the main technical characteristics of the Bluetooth Low Energy Personal-Area-Network technology. Our main reference is the Core Specification 5.0 available on the official website.

2.1 Network Topology and Devices

Devices can be divided into three categories: *Advertisers*, *Scanners* and *Initiators*. Communication has to be started by an initiator device following an *advertisement connectable packet* (`ADV_IND`). Advertisement is performed on the primary channels, while bidirectional communication happens on one of the 37 secondary channels, decided during the connection procedure. When a connection is established, the initiator and the advertiser respectively become the master and slave devices; while the former can be involved in more than one communication, the slave device can belong to a single *piconet* at a time. In fact, it is not possible to have a BLE device paired with more than one "master" device at the same time.

Each device's role is defined by the Generic Access Profile (GAP). As such, we can say that GAP contributes in shaping the topology of the network. Specifically, GAP defines the *Broadcaster*, *Observer*, *Peripheral* and *Central* roles. Broadcaster and Observer devices are, respectively, optimized for broadcast and reception; differently, Peripheral and Central devices work better in bidirectional communications.

As we had the possibility to concretely program a BLE device, we show in Listing 2.1 an extract of code in which the device is advertising *connectable* packets.

Listing 2.1: Set up flags for device mode of operation: connectable

```
ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::
    BREDR_NOT_SUPPORTED | GapAdvertisingData::
    LE_GENERAL_DISCOVERABLE);
ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::
    COMPLETE_LIST_16BIT_SERVICE_IDS, (uint8_t *)uuid16_list,
    sizeof(uuid16_list));
ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::
    THERMOMETER_EAR);
ble.gap().accumulateAdvertisingPayload(GapAdvertisingData::
    COMPLETE_LOCAL_NAME, (uint8_t *)DEVICE_NAME, sizeof(
    DEVICE_NAME));
ble.gap().setAdvertisingType(GapAdvertisingParams::
    ADV_CONNECTABLE_UNDIRECTED);
ble.gap().setAdvertisingInterval(1000); /* 1000ms. */
ble.gap().startAdvertising();
```

Advertising may also be *non-connectable* (`ADV_NONCONN_IND`): the LE device periodically sends its data on the main channel for every scanner to read. This is really convenient in terms of implementation, but it also represent the least secure solution, as data is sent in clear text. Moreover, since there is no acknowledgment nor response by the receiving party, this method is also the less reliable. In Listing 2.2 we highlight the different parameter used to set *non-connectable* advertising.

Listing 2.2: Set up flags for device mode of operation: non connectable

```
ble.gap().setAdvertisingType(GapAdvertisingParams::
    ADV_NON_CONNECTABLE_UNDIRECTED);
```

On the whole, the Specification defines six different advertising packets, among which we also report the request for additional information from the advertisement (`SCAN_REQ`), it having a direct corresponding bash command. Advertisement is performed at random intervals to avoid collisions, thus it may happen that the target device is not immediately discovered by the scanner.

2.2 Connection

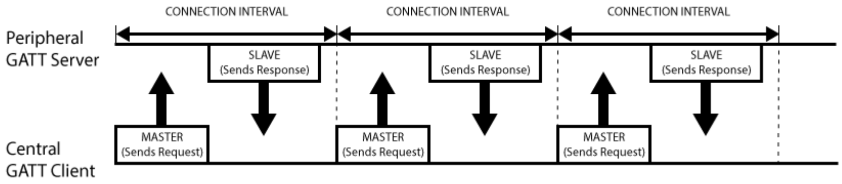
When the Scanner device answers to an advertisement message, a secondary channel is established for bidirectional communication. This channel is randomly chosen and communicated to the advertiser in the connection request packet. As previously mentioned, the slave device can be paired with a single master at a time, consequently all subsequent communications will happen in the established secondary channel, also granting additional connection speed.

Pairing procedures vary depending on the Bluetooth Low Energy device. They are mentioned in the Specification as *Association modes* and they are the following:

- Numeric Comparison: used when the devices can show numbers (at least 6 digits) and can receive user input (e.g. yes or no), like a phone or a laptop. Devices wishing to pair should show the same number, and the owners have to confirm the match.
    - This usually happen when pairing two mobile phones or a mobile phone to a laptop. In case the BLE device has a display, this association mode is attempted.
- Just Works: used when 1+ devices doesn't have a keyboard for user IO and / or cannot display 6 digits. Consequently, no PIN is shown and the user just has to accept the connection.
    - This is the case of the STM IoT Node. When attempting to connect from any Scanner device, the connection is established without authentication.
    - It is also the case of the Magic Blue smart bulb: the device pairs with any scanner.
    - In both situations, to unpair from the device it is necessary to unplug from the power source.
- Out Of Band: used when there is a (secure) OOB channel for pairing and security keys exchange. Of course, if such channel is not secure the whole process may be compromised.
- Passkey Entry: one device has input capabilities, but can't display 6 digits; the other device has output capabilities: the PIN is showed on the second device, and the connection is "confirmed" by entering the same PIN on the first device. Note that in this case the PIN is created by a specific security algorithm, while in legacy versions was an input from the user.
    - This is the case of the Mi Band 2 smart band, in which it is not possible to display PIN numbers, but there is an input device: when attempting to pair, the band vibrates and asks for confirmation by tapping its button.

2.3 GATT Transactions

While GAP defines how BLE-enabled devices can make themselves available, GATT (Generic ATTribute Profile) defines in detail how two Bluetooth Low Energy devices can connect and transfer data back and forth. The communication features *Profiles*, *Services* and *Characteristics*: they make use of the Attribute Protocol (ATT) that stores all the details related to the device



in a simple lookup table, using 16-bit IDs for each entry. Once the advertising process governed by GAP has concluded, it's the GATT turn to enter the scene. The established connection is not symmetrical: the Peripheral can connect to a single Central while the Central can connect with multiple Peripheral devices. A bidirectional connection is the only way to share data between devices, although it may also be possible for Peripherals to exchange data between themselves through a mailbox system; this architecture has a central unit for message dispatch, but has to be manually implemented.

GATT basically supports a server/client relationship where the entities are called GATT Server and GATT Client, the latter being usually a phone or a tablet that sendings requests to the server. The master device is responsible for the initiation of the transaction, and once the connection is stable, the Peripheral will suggest a *Connection Interval* for the connection, to see if there is new data available. However it is not compulsory for the central device to honour the request of the client if resources are not available.

2.3.1 Profiles, Services and Characteristics

As previously mentioned, GATT transactions make use of high-level nested objects called Profiles, Services and Characteristics.

- Profile: it is the collection of Services on the device that has been compiled by the peripheral designers.
- Services: they contain specific chunks of data that are the Characteristics. A service can have one or more characteristics and each service is identified through a unique number called UUID, that can be either 16-bit or 128-bit depending on the manufacturing of the device. Services can be explored on the Services of the Bluetooth Developer Portal, thus it's easy to track a service and its purpose in case the Service is named "Unknown".
- Characteristics: this is the lowest level in a GATT transaction and contains a single data point (it may be a single value or an array of bytes, depending on the type of data transmitted). Characteristics are composed by various elements such as a type, a value, properties and permissions. Properties, in particular, define what another device can do with the characteristics over Bluetooth in terms of operations such as READ, WRITE or NOTIFY.

  - READING a characteristic means copying the current value to the connected device.
  - WRITING allows the connected device to insert new values of data.
  - NOTIFICATIONS consist in a message type periodically sent in case of changes in a value.

  Permissions specify the condition that must be met before reading or writing data to the characteristic is granted.

- Descriptors: there is another level below the Characteristics level, that contains meta-data related to it. For example it's possible enable or disable notifications through the Client Characteristic Configuration Descriptor.

Chapter 3

Hacking BLE devices

This chapter is dedicated to BLE traffic analysis between two BLE devices. First, we will show how we analysed, reverse-engineered and hacked a Magic Blue smart bulb, with the aid of a mobile phone. It is possible to download the Android application which works as an interface with the BLE device; we also used Wireshark 2.6.2 to easily process captured traffic.This tool is unable to sniff BLE packets unless another support is provided (like Ubertooth or Nordic Semiconductor dongle), so we decided to use Android Developer Tools to register incoming and outgoing Bluetooth traffic. Through this setting, it becomes easy to track all the packets from the phone to the smart bulb in a single log file.

In the second example, we developed our own code on the Mbed Online Compiler and tested it on the STM IoT Node. We then tried to connect to the board from different interfaces (as our laptop's or mobile phone's) and used Wireshark to capture and interpret the traffic.

3.1 Case study: Smart Bulb

3.1.1 Connect to the device

The first step in order to sniff packets from a device requires to know its MAC address and understand its profile is structure. There are many open-source tools to accomplish this task, e.g. `hcitool`, `bluetoothctl` and `gatttool`. They come within the Bluez stack and can be easily used from the command line, where Bluez is the official Linux Bluetooth Protocol and provides support for Bluetooth Layers and Protocols. Due to its modularity, it is easy to install the needed modules and libraries, and quickly get to work with devices. In Listings 3.1, 3.2, 3.3 we show some possible usages of these three tools.

Among other possibilities, it is worth mentioning a widespread Python implementation called `bluepy`, which provides an interface for the same functionalities and allows constructing more complex programs.

In conclusion, they all provide similar functions and the user can choose depending on her specific requirements and targets. In this project, the main objective was to scan the surroundings for BLE and BT devices, connecting to them and gathering information about their services and characteristics.

Following are the command lines to scan and connect to the devices.

Listing 3.1: Connect to device via `hcitool`.

```
root@kali:~# hcitool lescan
LE Scan ...
D5:7A:0D:25:DE:50 (unknown)
D5:7A:0D:25:DE:50 MI Band 2
77:4E:A5:A1:B9:28 (unknown)
```

Listing 3.2: Connect to device via `bluetoothctl`.

```
chiara@chiara-N56VZ:~$ bluetoothctl
[NEW] Controller 6C:71:D9:2C:3B:2E chiara-N56VZ [default]
[NEW] Device F8:1D:78:63:3D:FA LEDBLE-78633DFA

[bluetooth]# scan on
Discovery started
Device F8:1D:78:63:3D:FA LEDBLE-78633DFA
Device 1C:B7:2C:56:F6:88 ASUS_Z00AD
Device 4B:17:24:EA:C4:87 4B-17-24-EA-C4-87
Device 68:64:4B:01:77:93 68-64-4B-01-77-93

[bluetooth]# connect F8:1D:78:63:3D:FA
Attempting to connect with F8:1D:78:63:3D:FA
[CHG] Device F8:1D:78:63:3D:FA Connected: yes
```

Listing 3.3: Connect to device via `gatttool`.

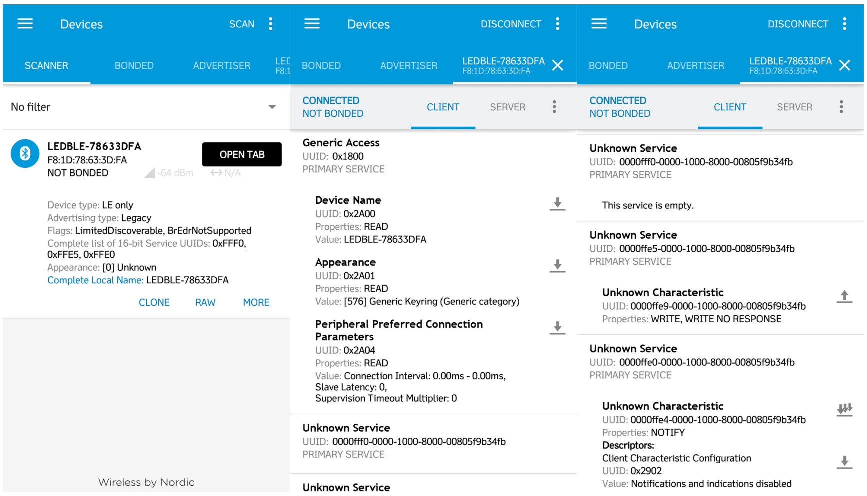
```
chiara@chiara-N56VZ:~$ gatttool -I

[                    ][LE]> connect F8:1D:78:63:3D:FA
Attempting to connect to F8:1D:78:63:3D:FA
Connection successful
```

We chose to use `gatttool` due to its efficiency in displaying the characteristics and services of the target device. Consequently, we will only consider this tool in further project developments and code excerpts.

A connection can be established simply via the `connect` keyword; afterwards it is possible to explore the characteristics (all of them or only the primary ones) of the device. We have reported the output of the connection process, together with an example of primary services for the smart bulb (see Appendix B). However, as the code shows, it is impossible to understand the meaning and purpose of each characteristic, and no support is provided by the Bluetooth GATT Services table.

We tested also an alternative way to connect to the smart bulb, employing an Android app as BLEScanner or nRFConnect. To proceed with



LEDBLE-78633DFA
F8:1D:78:63:3D:FA
NOT BONDED -64 dBm ↔ N/A

Device type: LE only
Advertising type: Legacy
Flags: LimitedDiscoverable, BrEdrNotSupported
Complete list of 16-bit Service UUIDs: 0xFFF0, 0xFFE5, 0xFFE0
Appearance: [0] Unknown
Complete Local Name: LEDBLE-78633DFA

LEDBLE-78633DFA
F8:1D:78:63:3D:FA

Generic Access
UUID: 0x1800
PRIMARY SERVICE

Device Name
UUID: 0x2A00
Properties: READ
Value: LEDBLE-78633DFA

Appearance
UUID: 0x2A01
Properties: READ
Value: [576] Generic Keyring (Generic category)

Peripheral Preferred Connection Parameters
UUID: 0x2A04
Properties: READ
Value: Connection Interval: 0.00ms - 0.00ms,
Slave Latency: 0,
Supervision Timeout Multiplier: 0

Unknown Service
UUID: 0000fff0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

Unknown Service

LEDBLE-78633DFA
F8:1D:78:63:3D:FA

Unknown Service
UUID: 0000fff0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

This service is empty.

Unknown Service
UUID: 0000ffe5-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

Unknown Characteristic
UUID: 0000ffe9-0000-1000-8000-00805f9b34fb
Properties: WRITE, WRITE NO RESPONSE

Unknown Service
UUID: 0000ffe0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

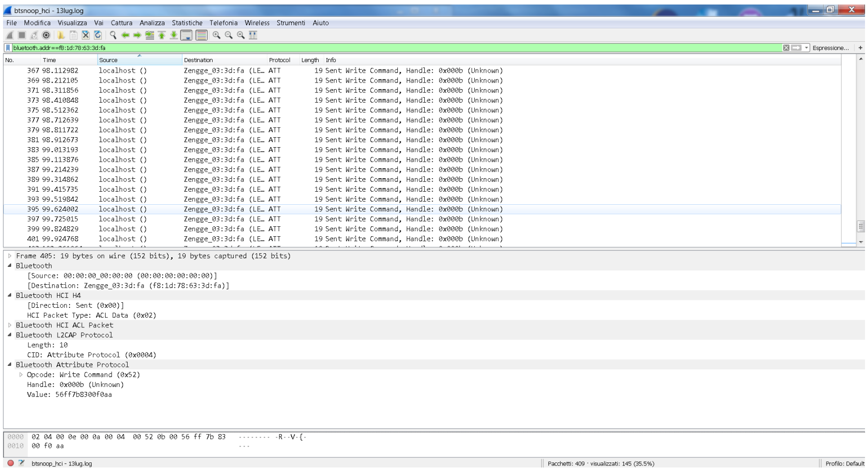
Unknown Characteristic
UUID: 0000ffe4-0000-1000-8000-00805f9b34fb
Properties: NOTIFY
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Notifications and indications disabled

Figure 3.1: Screenshots from the nRFConnect Android app when connecting to the smart bulb.

this approach, turn the Bluetooth on, and in the main page connectible devices will appear: the user can then easily open another tab and explore the content more deeply.

Although all the services are labelled as *unknown* and it is not clear what their values mean, the structure of the device profile, including characteristics, services and descriptors is apparent. It becomes now clear that knowledge of the characteristics and services of a device is not sufficient to understand the meaning of the exchanged data. We thus moved to a different approach: reverse-engineering of the packet structure and content, so to finally understand its meaning. Unfortunately, as previously mentioned, due to the lack of a sniffing device (e.g. a dongle) we decided to use a "legit" way to capture some traffic packets. From Android 4.0 onward, Developer Tools provide the user with a tool to capture all Bluetooth HCI packets in a file, called `HCI snoop log`. It is important to highlight that this section is not usually available and it has to be activated directly from the user.

In the next step we connect to the MagicBlue app and select the smart bulb from the list of available devices. After pairing, it is possible to choose from a wide range of colours and modes, to modify the brightness and a blinking time interval, as well as to enable the microphone sensitivity or play music from the smart phone and light up the bulb accordingly. Any of these operations requires the phone to send the bulb data packets containing the command requests for the selected function, and all this information is recorded in a snoop log file. After transferring the file from the phone to a PC, Wireshark is the instrument needed to parse the log file.


btsnoop_hci - 13lug.log
File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto
Espressione...
Pacchetti: 409 · visualizzati: 145 (35.5%)
Profilo: Default

```
bluetooth.addr==f8:1d:78:63:3d:fa
```

```
No.  Time       Source        Destination          Protocol Length Info
 367 98.112982  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 369 98.212105  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 371 98.311856  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 373 98.410848  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 375 98.512362  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 377 98.712639  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 379 98.811722  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 381 98.912673  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 383 99.013193  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 385 99.113876  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 387 99.214239  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 389 99.314862  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 391 99.415735  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 393 99.519842  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 395 99.624002  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 397 99.725015  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 399 99.824829  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
 401 99.924768  localhost ()  Zengge_03:3d:fa (LE… ATT      19 Sent Write Command, Handle: 0x000b (Unknown)
```

```
▷ Frame 405: 19 bytes on wire (152 bits), 19 bytes captured (152 bits)
◢ Bluetooth
    [Source: 00:00:00_00:00:00 (00:00:00:00:00:00)]
    [Destination: Zengge_03:3d:fa (f8:1d:78:63:3d:fa)]
◢ Bluetooth HCI H4
    [Direction: Sent (0x00)]
    HCI Packet Type: ACL Data (0x02)
▷ Bluetooth HCI ACL Packet
◢ Bluetooth L2CAP Protocol
    Length: 10
    CID: Attribute Protocol (0x0004)
◢ Bluetooth Attribute Protocol
  ▷ Opcode: Write Command (0x52)
    Handle: 0x000b (Unknown)
    Value: 56ff7b8300f0aa
```

```
0000  02 04 00 0e 00 0a 00 04  00 52 0b 00 56 ff 7b 83   ········ ·R··V·{·
0010  00 f0 aa                                            ···
```

Figure 3.2: Applying a filter in Wireshark and observing packets contents.

3.1.2 Traffic Analysis

One of the obstacles in understanding the structure of data packets exchanged between two BLE devices lies in finding the packets of interest. The amount of data gathered in the log file is indeed impressive, but many of the packets are related to the connection phase between the two entities and so they hold no value for our research. It is thus advisable to filter on the Bluetooth address of the target device, acquired during the initial scanning phase. Wireshark allows selecting it either via its specific toolbar or via right click, once a meaningful packet has been found. We then proceed to analyse the structure of the remaining packets, leveraging on Wireshark parser, which helps in detecting recurring similarities. We can now understand how the colour information is encoded inside the data structure.

We summarized our findings in the following key considerations.

- Within the Attribute Protocol, `Value` length can be up to a maximum of 20 bytes, depending on the information it transmits.
- Writeable Characteristics are usually marked with a ffe9-XX UUID, referring to a ffe5-XX Service.
- There are no security layers and it is possible to write any value.
- This can be easily accomplished via `gatttool` or `bluetoothctl` on the command line, or using a BLE sniffer smart phone app.

In tables 3.1 and 3.2 we provide a short summary of the patterns recognized in Magic Blue smart bulb traffic. The bulb has indeed many more functionalities than the ones reported here, which we did not further investigate.

Table 3.1: Single colour choice

Hex value	Meaning
56	Fixed prefix
ff085a	Colour RGB code
00	From 00 to ff, brightness of the light
0f OR f0	off/on
aa	Fixed suffix

Table 3.2: Light Functions

Hex value	Meaning
bb	Fixed prefix
3 OR 2	Respectively active functions *strobo* or *gradual change*
0	From 00 to ff, brightness of the light
1	On-time of the lightbulb, from a quick flash (1) to 5 seconds
f	Interval of time between flashes (1 faster than f)
44	Fixed suffix

Hacking a smart bulb may be also accomplished by a Python script compiled for a Raspberry Pi device. There is a single requirement on the operating system of the component, as it is needed to install Bluez components as well as a compiler. This solution would allow controlling the device remotely in a continuous way. Moreover, there exists a full Python Compiler called MicroPython for ST Microelectronics, though not compatible with the board used in this project. The list of supported boards is available from the official site.

3.2 Case Study: STM IoT Node

For this experiment we used the STM IoT node and tested it on both Linux distributions. The board retrieves the temperature and humidity levels in the environment and broadcasts them as a non-connectable advertising packet, as we can see in Listing 3.4.

Listing 3.4: Retrieve sensor values and update packet

```
uint8_t temp = static_cast<unsigned int>(sensor_value);
uint8_t hum = static_cast<unsigned int>(humidity_value);

uint8_t service_data[4];
service_data[0] = GAPButtonUUID & 0xff;
service_data[1] = GAPButtonUUID >> 8;
service_data[2] = temp;
service_data[3] = hum;

```

```
BLE::Instance().gap().updateAdvertisingPayload(
    GapAdvertisingData::SERVICE_DATA, (uint8_t *)service_data,
    sizeof(service_data));
```

Packets broadcasted by the board can be retrieved with any of the tools described at the end of the previous chapter. We analysed the payload and found that it follows this structure:

length - meaning - content.

The length is expressed in bytes and it refers to the overall size of *meaning* and *content*. The former two is a label which has to be matched with the corresponding Bluetooth GAP Assigned Number (specification available on the website, we report the table in Appendix A) to understand its meaning. Common examples are `0x09` for "Complete Local Name" or `0x0a` for "Power Level". In our case, the label corresponds to "Service data 16-bit UUID", which means that we should look at the next two bytes to have a more precise indication of the received data. Vendors usually publish a web page in which they list all their codes with the respective meaning, while in this example we can see that its value is added to the payload in `service_data[0]`, and corresponds to `00 aa`.

Finally, the next two bytes encode the temperature and humidity in the room.

Bibliography

Appendix A

GATT Assigned Numbers

Data Type	Data Type Name	See advlib section
0x01	Flags	Flags
0x02	Incomplete List of 16-bit UUIDs	UUID
0x03	Complete List of 16-bit UUIDs	UUID
0x04	Incomplete List of 32-bit UUIDs	UUID
0x05	Complete List of 32-bit UUIDs	UUID
0x06	Incomplete List of 128-bit UUIDs	UUID
0x07	Complete List of 128-bit UUIDs	UUID
0x08	Shortened Local Name	Local Name
0x09	Complete Local Name	Local Name
0x0a	Tx Power Level	Tx Power
0x0d	Class of Device	Generic Data
0x0e	Simple Pairing Hash C-192	Generic Data
0x0f	Simple Pairing Randomizer R-192	Generic Data
0x10	Security Manager TK Value	Generic Data
0x11	Security Manager OOB Flags	Generic Data
0x12	Slave Connection Interval Range	SCIR
0x14	16-bit Solicitation UUIDs	Solicitation
0x15	128-bit Solicitation UUIDs	Solicitation
0x16	Service Data 16-bit UUID	Service Data
0x17	Public Target Address	Generic Data
0x18	Random Target Address	Generic Data
0x19	Public Target Address	Generic Data
0x1a	Advertising Interval	Generic Data
0x1b	LE Bluetooth Device Address	Generic Data
0x1c	LE Bluetooth Role	Generic Data
0x1d	Simple Pairing Hash C-256	Generic Data
0x1e	Simple Pairing Hash Randomizer C-256	Generic Data
0x1f	32-bit Solicitation UUIDs	Solicitation

0x20	Service Data 32-bit UUID	Service Data
0x21	Service Data 128-bit UUID	Service Data
0x22	LE Secure Con. Confirmation Value	Generic Data
0x23	LE Secure Connections Random Value	Generic Data
0x24	URI	Generic Data
0x25	Indoor Positioning	Generic Data
0x26	Transport Discovery Data	Generic Data
0x27	LE Supported Features	Generic Data
0x28	Channel Map Update Indication	Generic Data
0x29	PB-ADV	Generic Data
0x2a	Mesh Message	Generic Data
0x2b	Mesh Beacon	Generic Data
0x3d	3-D Information Data	Generic Data
0xff	Manufacturer Specific Data	Mfr. Specific Data

Appendix B

Smart Bulb Services and Characteristics

Listing B.1: Smart bulb Primary Characteristics and Services.

```
[F8:1D:78:63:3D:FA][LE]> help
help                                           Show this help
exit                                           Exit interactive mode
quit                                           Exit interactive mode
connect         [address [address type]]       Connect to a remote device
disconnect                                     Disconnect from a remote device
primary         [UUID]                         Primary Service Discovery
included        [start hnd [end hnd]]          Find Included Services
characteristics [start hnd [end hnd [UUID]]]   Characteristics Discovery
char-desc       [start hnd] [end hnd]          Characteristics Descriptor Discovery
char-read-hnd   <handle>                       Characteristics Value/Descriptor Read by handle
char-read-uuid  <UUID> [start hnd] [end hnd]   Characteristics Value/Descriptor Read by UUID
char-write-req  <handle> <new value>           Characteristic Value Write (Write Request)
char-write-cmd  <handle> <new value>           Characteristic Value Write (No response)
sec-level       [low | medium | high]          Set security level. Default: low
mtu             <value>                        Exchange MTU for GATT/ATT

[F8:1D:78:63:3D:FA][LE]> characteristics
handle: 0x0002, char properties: 0x02, char value handle: 0x0003, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, char properties: 0x02, char value handle: 0x0005, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0006, char properties: 0x02, char value handle: 0x0007, uuid: 00002a04-0000-1000-8000-00805f9b34fb
handle: 0x000a, char properties: 0x0c, char value handle: 0x000b, uuid: 0000ffe9-0000-1000-8000-00805f9b34fb
```

```
handle: 0x000d, char properties: 0x10, char value handle: 0x000e, uuid: 0000ffe4-0000-1000-8000-00805f9b34fb

[F8:1D:78:63:3D:FA][LE]> primary
attr handle: 0x0001, end grp handle: 0x0007 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0008, end grp handle: 0x0008 uuid: 0000fff0-0000-1000-8000-00805f9b34fb
attr handle: 0x0009, end grp handle: 0x000b uuid: 0000ffe5-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0xffff uuid: 0000ffe0-0000-1000-8000-00805f9b34f
```