Reverse Engineering BLE Devices Documentation

Sergio Alberti

Sep 01, 2019

Contents

The following documentation is intended as a guide to reverse engineering of BLE (*Bluetooth Low Energy*) devices. The idea is to provide information about BLE, how to identify the protocol used by the devices and how to create shell scripts to communicate with them.

To do this, the guide is based on examples applied to devices currently on the market. As explained in the *Contributions* section, this document would like to be an evolving project, in which to gather information on reverse engineering techniques and to make available works already done in this area.

CHAPTER 1

Notes (GSoC 2018)

This guide comes from a project of GSoC 2018 and takes as a starting point a work done on radiator valves. These systems have become increasingly important in recent years, especially in some countries where they have been made mandatory by law. This led to the production of various models programmable using a smartphone application coupled with the BLE protocol. At the moment all the products on the market use proprietary communication protocols to exchange essential data with the application, making it difficult to integrate this devices into external open-source projects. For this reason the University of Milan has successfully reverse-engineered a protocol and released the necessary code to use it with a GPL license. An English translation of the code can be found here.

The project aims to use what has already been produced to:

- write a reverse-engineering guide for BLE devices as general as possible
- design a mechanical device to test the valves without a radiator
- port the library to a more modern language in an attempt to integrate it into projects such as openhab or home-assistant and create a Debian package

Here **is available a detailed description of the deliverables and the time schedule and** here **is a brief weekly report.**

CHAPTER 2

Contents

2.1 Introduction

For some years now the world of IoT (*Internet of Things*) has experienced a strong increase in the production of BLE (*Bluetooth Low Energy*) devices. This type of "smart devices" changes the way we interact with the world, but is often controlled through smartphone apps or software whose application protocol is not disclosed.

This guide deals with the activity related to the reverse-engineering of the protocol used in the communication with a BLE device.

It tries to be as general as possible, however it takes as a test/reference device the Eqiva radiator valves produced by the EQ3 company and the related Android application CalorBT . Between 2015 and 2016, Dr. Andrea Trentini contacted the company more than once in order to obtain documentation related to the protocol, but the company did not want to provide details. EQ3 also specified that there is no GNU/Linux software to interact with the valves.

The objectives are:

- to reverse engineer the protocol used to communicate with the BLE device
- to show how to communicate with the device using the BlueZ stack

The result translates into the possibility to integrate these devices into free home automation systems or other external projects. In the specific case of the radiator valves, this guide has led to the creation of a series of shell script functions to manage every "device controlling" aspect.

Fig. 1: EQ3 Eqiva valve on a radiator

2.1.1 What Radiator Valves Are

Radiator valves are thermoregulation devices composed of two parts: a valve and a bulb in contact with the surrounding environment. The bulb contains a fluid with a high expansion coefficient. The set of these two components allows to create a system that, according to a range of values placed on the valve, expands or contracts the fluid. This causes the activation or the interruption of the flow to be managed. It is therefore clear that the application area is that of heating and cooling systems. The use of thermostatic valves on domestic heaters has become fundamental because it allows the reduction of consumption and emissions.

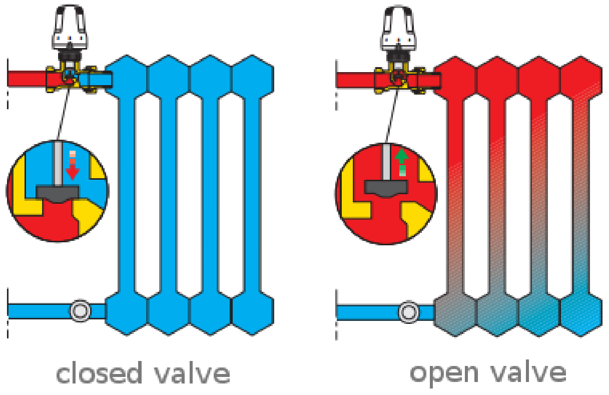


Fig. 2: Operating scheme of a classic valve[1]

Electronic versions of these valves have been available for some years now. They are battery pow-

[1] Demshop - Caleffi 200

ered and equipped with an integrated thermostat replacing the classic fluid. Theese valves can be programmed for the whole week and allow a more accurate selection of the desired temperature. For ease of use, the interaction with these valves usually takes place through a proprietary smartphone application that uses a Bluetooth or Wi-Fi connection for data exchange.

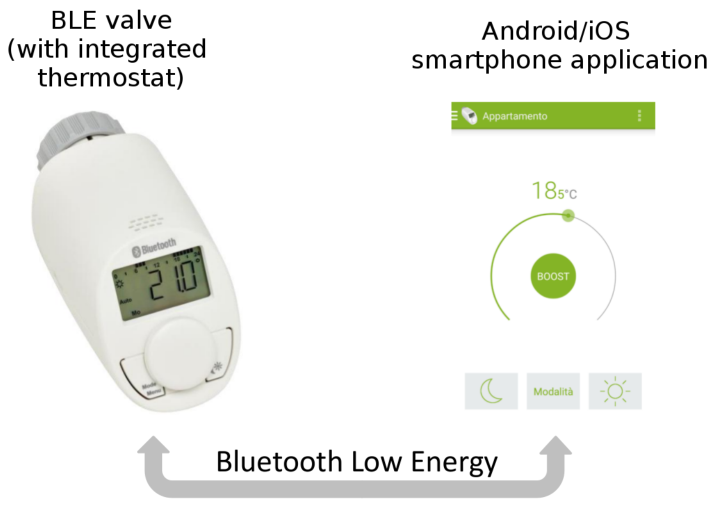


Fig. 3: Electronic BLE valve

2.1.2 Setup Without A Radiator

Once mounted on the radiator, the valves must perform an *adaptive run* phase before they can be used. During this phase the valve lengthens the external pin until it detects the radiator's pin and calibrates itself (on the strength of the valve pin). While this activity takes place, the `AdA` message is shown on the display and no further operations can be performed.

Note: According to the manual[2], if during the setup phase the valve shows the error messages `F2` or `F3` the reasons are respectively:

- the valve pin extended to the maximum length without meeting the radiator pin
- the valve pin was blocked before reaching the minimum required distance

While the error is shown, the motor returns to its starting position (indicated by `Ins` on the display). To restart the calibration, press the button in the middle of the valve and wait again for the necessary time.

However, it may happen that there isn't a radiator suitable for initialization or an immediate way to perform this phase. Several times we have done the calibration by inserting a marker in the valve

[2] Eq3 Eqiva User Manual

and locking it against a wall to exert the necessary force. It is easy to understand that this is *not* a very convenient method and often requires more attempts.

Since the valve must recognize that it is in contact with something *very constantly resistant*, alternative methods can be found. The least expensive solution involves using the **adapter** supplied with the valves. It is designed to make them usable on various types of radiators and is equipped with a screw to tighten and loosen the grip (see *Adapter Details*).

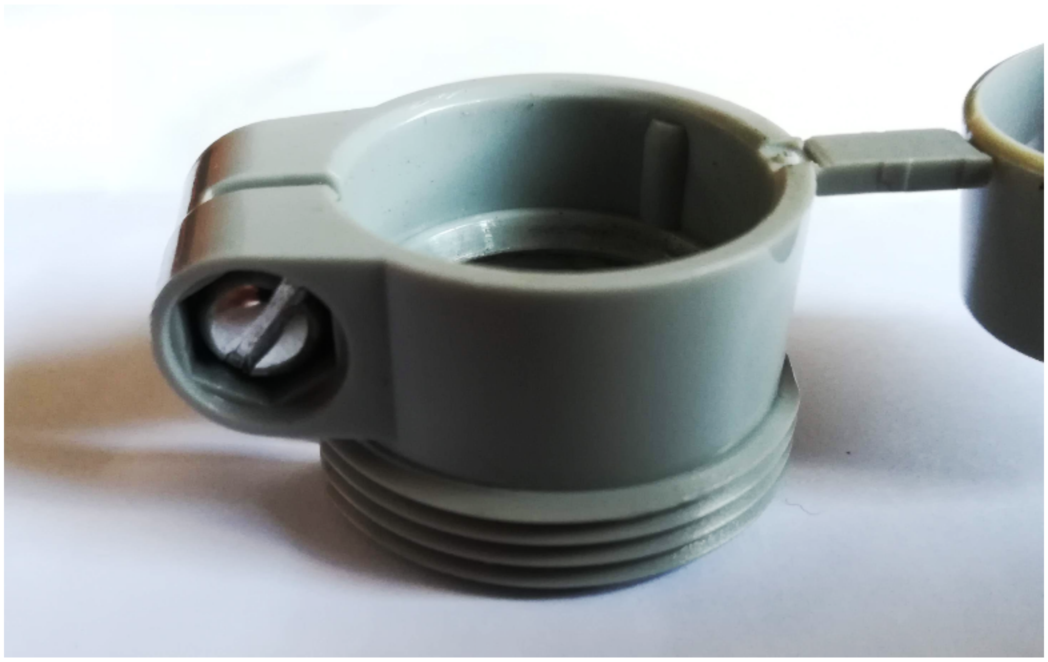


Fig. 4: Adapter Details

The second piece we need is a **large screw** (or a sort of cylinder) of the size needed to **fill the adapter**. (see the *next image* as a reference). It is important that when the adapter is tightened the screw makes a lot of grip because the valve's pin pushes very hard.

The bolt can be found in many hardware stores at low cost (*less than 1€*), while the adapter will cost you more (*between 5€ and 10€*). However it is usually included in the valve package.

Now it is enough to:

1. insert the screw in the adapter so that it sticks out a lot towards the valve
2. tighten the small screw around the adapter
3. attach the adapter to the valve (see *Test device setup example*)
4. start the setup phase by pressing the big button on the valve

If during the calibration phase we meet one of the *errors* mentioned in the *previous note*, then it's sufficient to **adjust the screw** accordingly.

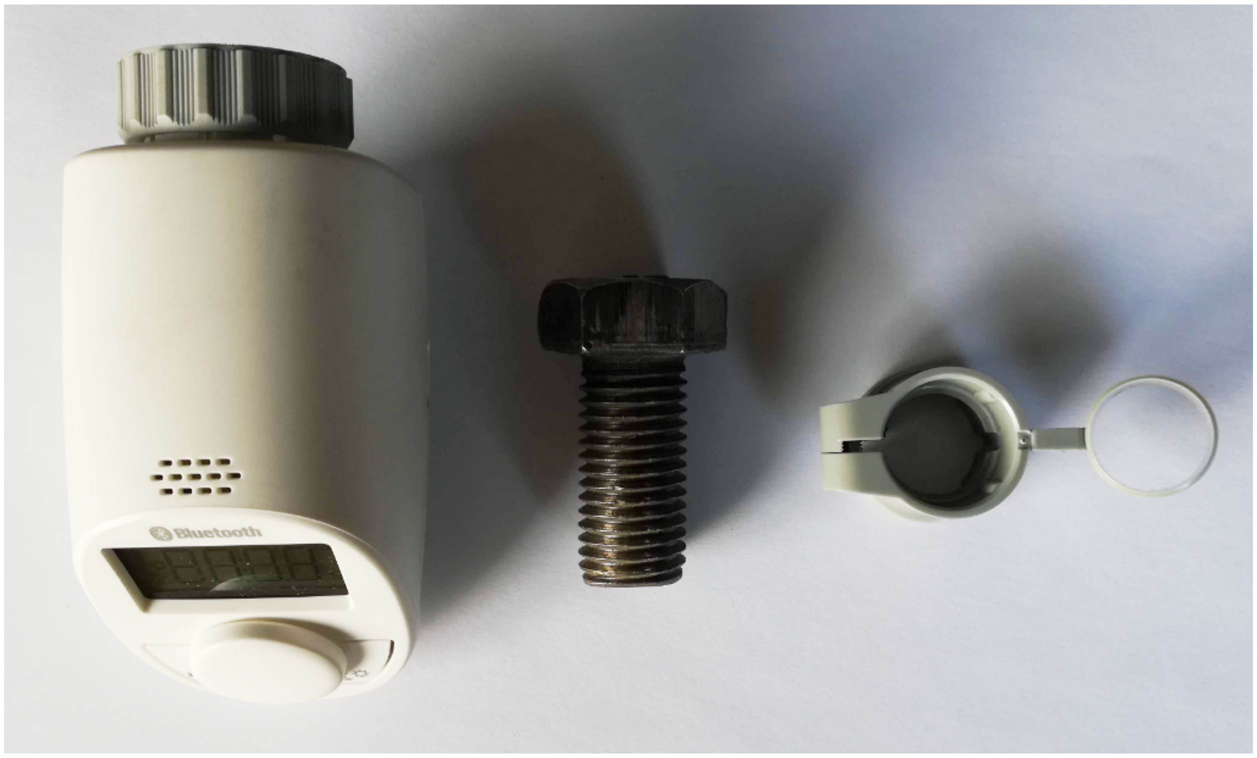


Fig. 5: From *left to right*: the valve, the screw and the adapter

Fig. 6: Test device setup example

2.2 Application Protocol Reverse Engineering

Bluetooth Low Energy (*BLE* or *Bluetooth Smart*) technology was born as a personal project of the Finnish company Nokia and only in 2010 was introduced in the Bluetooth 4.0 specification.

BLE has gained importance in the Internet Of Things (*IoT*) because it wants to ensure low energy consumption while maintaining a good range of communication. This is the reason why the main producers of mobile and desktop OSes provide complete support, allowing to design devices able to communicate with all the most modern platforms.

The reverse engineering work is carried out through two parallel actions:

- Logging and inspection of Bluetooth packages exchanged between the smartphone and the BLE device
- Decompilation of the Android application

This allows to understand the protocol used and exploit the application code to verify its correctness, also to ensure greater consistency with respect to the original specifications.

Note: This guide requires that you have an Android device available. However, there are advantages in using an emulator. If you are interested in this, take a look at the *Logging With An Emulator* section.

2.2.1 BLE: Operating Principles

The connection and transmission of data between two devices requires multiple steps and involves multiple elements[1]. The most important high-level components are listed and discussed below.

GAP (Generic Access Profile)

In order to notify the presence of a BLE device to the outside world, a process called *advertising* is necessary. It basically consists of constantly sending informative packets to devices enabled to use Bluetooth within a certain distance.

What manages the aspects related to the connection, to the advertising and finally determines whether two devices can interact with each other is the GAP, acronym of Generic Access Profile ([2] and[3]).

The division into **roles** is fundamental. We will distinguish:

- **Peripheral devices** (or just *peripheral*), with few resources and extremely variable nature

[1] Bluetooth Core Specification 5.0, Volume 1, Part A, Section 1.2
[2] Bluetooth Core Specification 5.0, Volume 1, Part A, Section 6.2
[3] Bluetooth Core Specification 5.0, Volume 3, Part C, Page 1966

- **Central devices**, such as smartphones, tablets and computers with a lot of computing power and memory

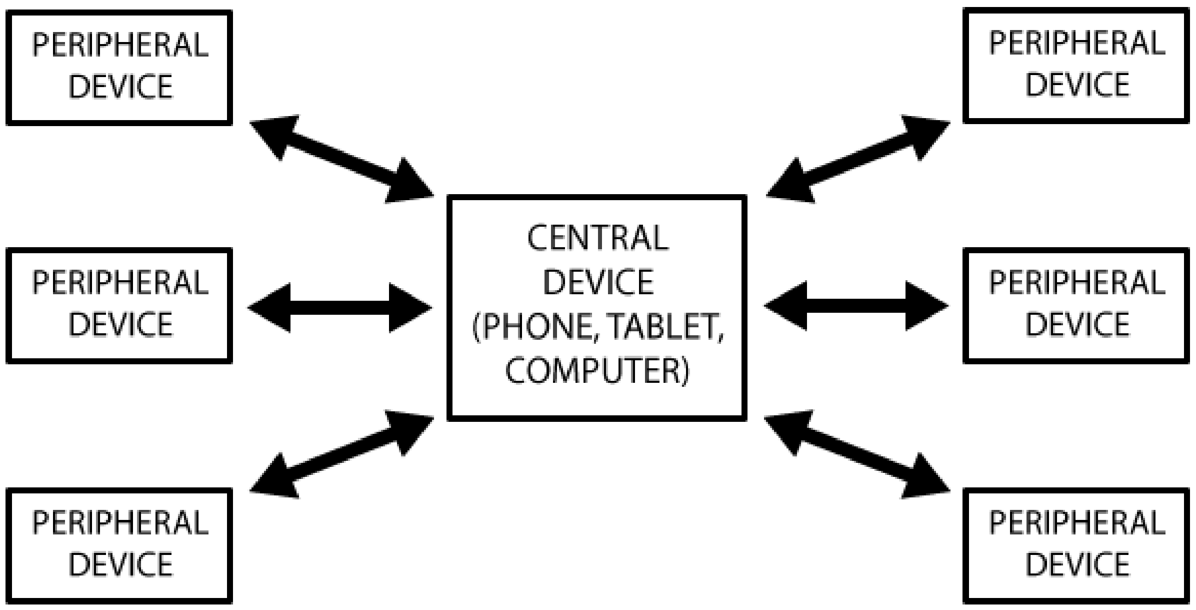


Fig. 7: Topology[5]

Peripheral devices cannot be connected to one another, they only communicate with central device one at a time (see *Topology 5*). Therefore, when a connection is established, they will block the advertising process until the connection is terminated.

In contrast, central devices can simultaneously manage data exchange with multiple devices. Consequently, the communication between two peripheral devices requires the creation of a special system that exploits this possibility.

Usually, the classic BLE devices in IoT (valves, lamps, scales and so on) fall into the type of *peripheral devices* while the application that manages them is installed on a *central device*. This is also what respectively happen with the radiator valves and the CalorBT application that we are considering.

GATT (Generic Attribute Profile)

When the connection is established, the bidirectional transmission takes place through the ATT protocol (*Attribute Protocol*) and uses the concepts of GATT profile, service and characteristic[4], which will be discussed soon.

[5] Kevin Townsend. Introduction to bluetooth low energy. A basic overview of key concepts for BLE.
[4] Bluetooth Specifications - GATT Overview

A significant aspect is given by the relationship that is created between the peripheral device and the central device. The former is referred to as **GATT Server** (or Slave) as it provides services and characteristics, while the latter is called **GATT Client** (or Master).

All transactions start from the Master and receive a response from the Slave which, at the time of the first connection, suggests a connection interval. At the end of each interval the Master reconnects to check the availability of new data. This is only a suggestion provided by the peripheral, which however does not place time constraints on the central device.

As already mentioned, GATT transactions are based on hierarchical high-level objects: profiles, services and characteristics (see *Hierarchy of profiles, services and characteristics 5*).

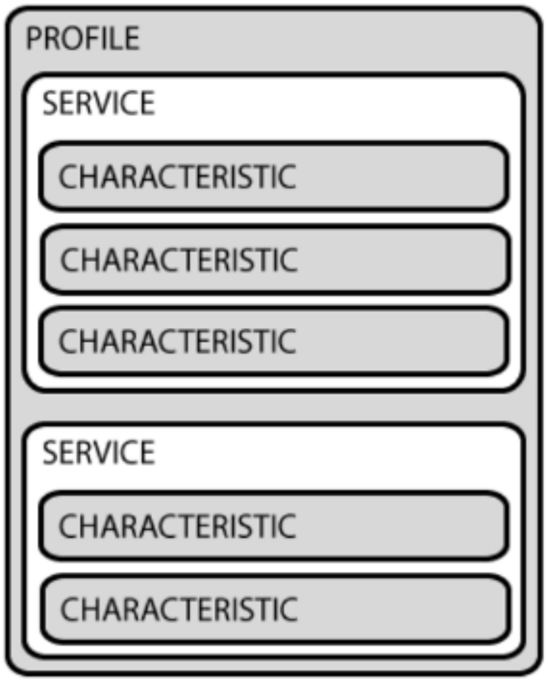


Fig. 8: Hierarchy of profiles, services and characteristics[5]

Profiles

Profiles define possible applications of the device, describing its functionality and use cases. The BLE specification provides a wide range of standard profiles that are used in various fields, but also allows manufacturers to create new profiles using GATT. This facilitates the development of innovative applications that still maintain interoperability with other Bluetooth devices.

For example, the "Blood Pressure Profile" and the "Proximity Profile" are *predefined* profiles. They are designed to be implemented by a blood pressure meter and to monitor the distance between two devices.

Services

Services allow to perform a first and not very detailed logical division of the data. They are composed of one ore more characteristics and are identified through a *UUID* consisting of:

- **16 bit** for the predefined services
- **128 bit** for those created specifically by peripheral devices manufacturers

For example, the aforementioned “Blood Pressure Profile” provides the services “Blood Pressure Service” and “Device Information Service”, necessary for the transmission of data about blood pressure and device status.

Characteristics

Characteristics are the main interaction point with the BLE peripheral and represent the most granular layer in the logical data division. Each characteristic handles information related to a single aspect, dealing with the transmission in one or both directions. For this reason they will be given **properties** such as *Read* or *Write*. Like the services, they are also identified through 16 or 128 bit UUIDs.

For example, the “Blood Pressure Measurement” characteristic provided by the “Blood Pressure Service” can be used to read the values measured by the blood pressure meter.

Notifications

As already mentioned, it is usually the GATT Client (*the Central device*) that initiates a transaction. However, even if they are not represented in the hierarchy, the BLE provides **Notifications** and **Indications** so that the GATT Server (*the Peripheral device*) can request for data or simply send information to the counterpart with or without an explicit signal from the latter.

In general, notifications are used to inform the client about the value assumed by a characteristic. For this reason, they are one of the possible values that can be assigned to the properties of a characteristic, together with the already mentioned Read and Write.

In order for the mechanism to work, an explicit request to receive notifications from the client is required.

To better clarify these concepts, consider that one of the characteristics defined by the manufacturers of the radiator valves *EQ3 Eqiva* has the following specifications:

- UUID: d0e8434d-cd29-0996-af41-6c90f4e0eb2a
- Property: read/write/notify

The 128-bit UUID allows us to understand that the characteristic has been defined by the producers and its properties tell us what operations we can perform on it.

2.2.2 BT/BLE: Main Differences

This section is not essential to understand the rest of *this* guide, but it is still useful to better figure out what is detected through the *logging activity*.

The next paragraphs explain the main differences between the classic Bluetooth implementation (Bluetooth BR/EDR) and the Bluetooth Low Energy. The goal is to deepen some details related to aspects of connection and consumption. Receiving and transmitting data requires a lot of energy and consequently interesting solutions have been studied to optimize these activities for some use cases.

Important aspects of Bluetooth BR/EDR[15]:

- transmits all **types of data** (including audio/video streams), ensuring high throughput
- **requires pairing**
- allows the use of different topologies (Piconet and Scatternet)

Important aspects of Bluetooth Low Energy[15]:

- **asynchronous data exchanges**, with low throughput (no streams)
- **optional pairing**
- basically uses Piconet topology
- low consumption: allows to power devices with *coin cell batteries* and still last over time

Just above the physical layer, which we will not cover, there are two very different **Data Link** layers. Both define a series of states in which a *single device* can be. The *next image* immediately highlights how Bluetooth BR/EDR devices are more complex.

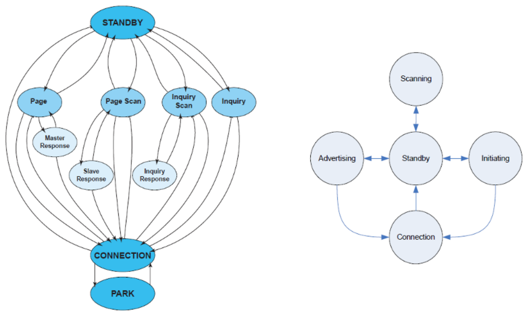


Fig. 9: *BR/EDR* States[12] (left) // *BLE* States[13] (right)

Observing the `Connection` states in the two state machines, we can see the *connection oriented* nature of the Bluetooth BR/EDR. Once the connection is established, the slave device can reduce its consumption by exploiting some **substates** (*Sniff* or *Hold* mode) or by entering in *Park mode* (see *BR/EDR Connected Slave Substates*). These alternatives keep the connection to the master device active, although *Park Mode* hides the device from the network.

In the case of BLE, the only way to save energy is to enter the `Standby` state. This however leads the device to lose the connection and restart the `Advertising` or `Initiating` phases (respectively for *Master* and *Slave*), creating a continuous sequence of *standby-search-connection*

[15] Ten Important Differences Between Bluetooth BREDR And Bluetooth Smart
[12] Confronto Tra Bluetooth Basic Rate e Bluetooth Low Energy
[13] How BLE works

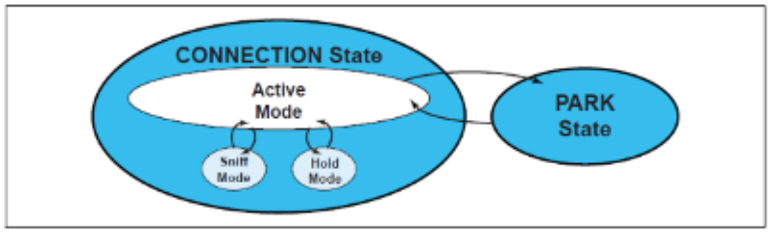


Fig. 10: Bluetooth *BR/EDR* Connected Slave Substates[12]

phases. The reason why repeating this "steps" allows you to consume few energy and maintain low response times lies in various factors.

First of all, BLE technology uses only **3 physical search channels** (*BR/EDR* uses 32 channels). Given the time required for packet transfer, this phase requires between `0.6ms` and `1.2ms` (while `22.5ms` in *BR/EDR*). This leads to a power saving of 10-20 times compared to the classic Bluetooth[14] .

Note: Given these different "architectures":

- using Bluetooth *BR/EDR* a master can connect with **up to 7 slaves** in active mode and 255 in park mode
- using BLE there are **no theoretical limitations** on the number of slaves to which a master can be connected

---

Still within the Data Link level, substantial differences are present in the transmission of packets:

- In Bluetooth BR/EDR each communication channel is divided into **slots of 625us** used alternately by the master and the slave. They can transmit a packet per slot. However, sending a package can take up to *5 slots*.
- In BLE the time units are **Events**. They vary in length depending on the decisions of the master (for *connection events*) and the advertiser (for *advertising events*).

While the Bluetooth BR/EDR uses a very strict transmission method, the one used by BLE is more flexible and can be optimized according to various parameters.

As an example, for Connections Events, the BLE specification provides a `connInterval`[16] value which indicates the minimum time that must elapse between two consecutive events of this type (between *7.5ms* and *4s*). Another parameter, `connSlaveLatency`[16], defines the number of Connection Events in which the slave is not forced to listen to the master and can stay in standby. This parameters are responsible for **consumption and latency times** and exist also for the Advertising Events.

[14] One Small Step For Bluetooth Low Energy Technology
[16] Bluetooth Core Specification 5.0, Volume 6, Part B, Page 2638

By increasing `connSlaveLatency` and keeping `connInterval` low, you can guarantee excellent consumption without lengthening latency times too much.

As already mentioned, the BR/EDR standard is designed to transmit any type of information, while the BLE prefers a few data at a time. The result is that in the first case several types of **logical transport** are defined: SCO and eSCO for synchronous communication, ACL for the asynchronous one and two types of Broadcast. The BLE alternative instead implements **only asynchronous ACL communication**.

This leads to a big difference in package format. As the *next image* shows, **BLE packages are shorter** (max 376 bit vs 2871 bit) and therefore require less transmission time. This was achieved by removing redundant information and limiting the payload size.

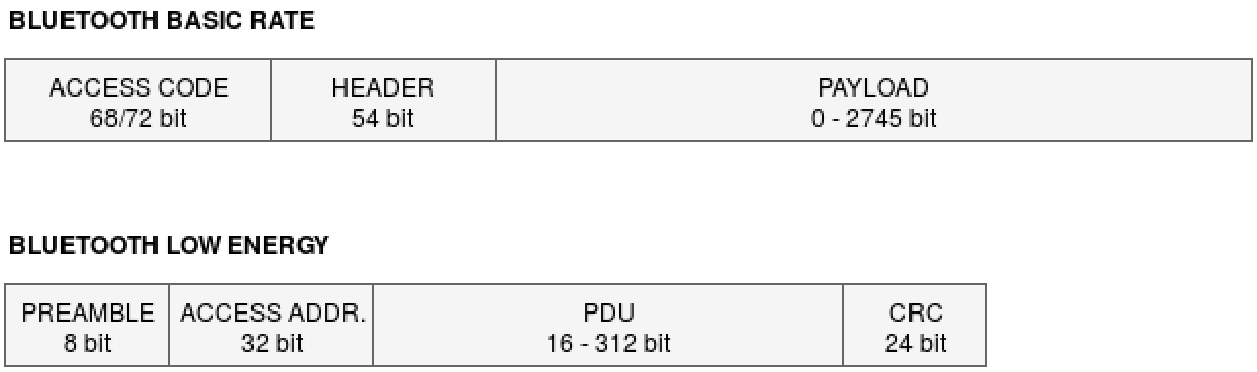


Fig. 11: *Bluetooth BR* and *BLE* packet format

By taking advantage of all these design choices, BLE can complete a connection (scan for devices, link, send data, authenticate, and go back to a standby state) in just `3ms`. The same activity with Bluetooth BR/EDR takes *hundreds* of milliseconds[14].

Reduced times lead to lower energy consumption and lower latency.

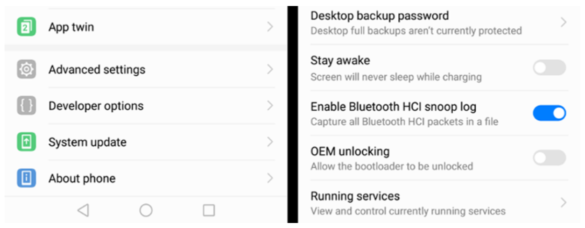
2.2.3 Logging Via Android

Logging is the activity of recording data and information related to certain operations as they are carried out. In this specific case it is a matter of tracing all the Bluetooth packages exchanged during the communication between a BLE device and a smartphone in order to inspect their contents.

From version *4.4 "KitKat"*, Android introduces the possibility to perform the logging of the packets sent and received via Bluetooth through the function **"Enable HCI Bluetooth snoop log"** in the "Developer Options" section.

Note: If there is no "Developer Options" entry in the settings (see *Android settings*), you can activate it by entering the "About Phone" section and clicking repeatedly (at least 8 times) on

Build Number or Version Number (depending on the version of Android) up to the appearance of a notification.



Desktop backup password
Desktop full backups aren't currently protected

Stay awake
Screen will never sleep while charging

Enable Bluetooth HCI snoop log
Capture all Bluetooth HCI packets in a file

OEM unlocking
Allow the bootloader to be unlocked

Running services
View and control currently running services

Fig. 12: Android settings

Once activated, the system starts to populate a file called *bootsnoop_hci.log*, which is usally placed inside the root directory of the smartphone (something like `/sdcard/`). The format is compatible with many protocol-analysis software.

Note: Unexpectedly, the acquisition of Bluetooth packages via Android **does not** require the smartphone to be rooted. Despite this, some manufacturers (e.g. *Huawei*) by default save the `bootsnoop_hci.log` file in a non-accessible directory or disable its generation. If you cannot find the log file, you're probably in one of this cases. It's therefore necessary to root the smartphone. Once that is done, within the Android system folders you can find a `bt_stack.conf` file (in my case in `/system/vendor/etc/bluetooth/`). Inside this file you can activate the generation of the log file and choose where to save it by appropriately modifying the following lines:

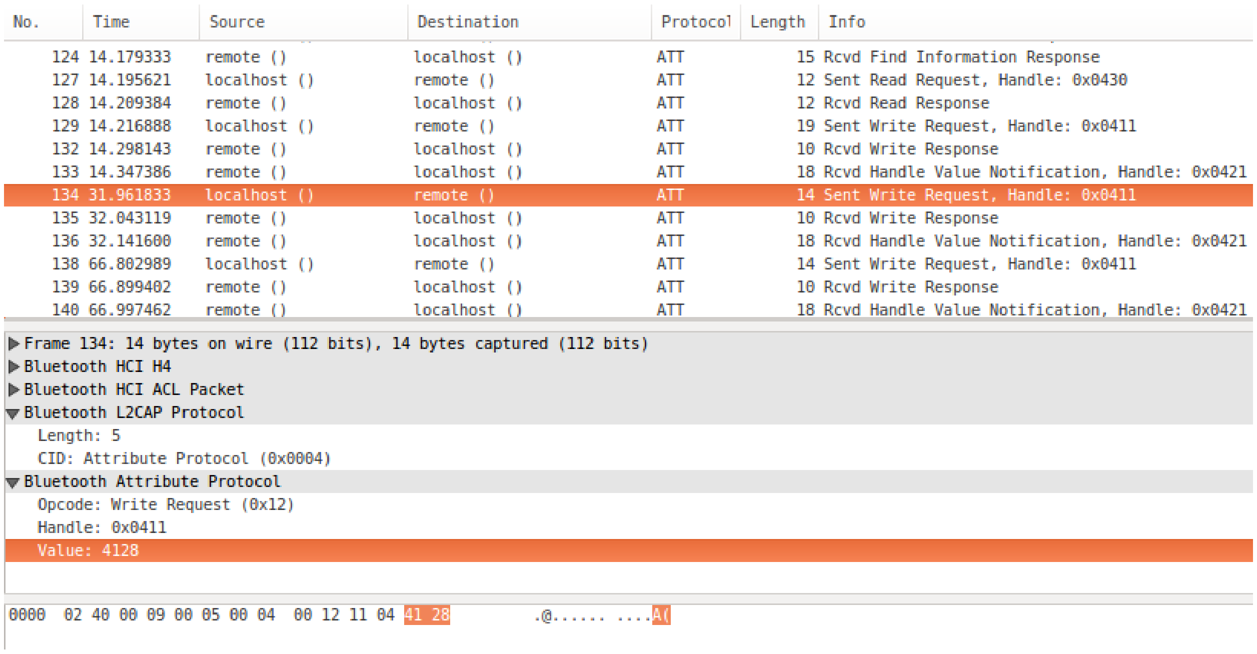
```
BtSnoopLogOutput=true
BtSnoopFileName=/data/log/bt/btsnoop_hci.log
```

The inspection software chosen is Wireshark, a free and opensource network protocol analyzer. However, the analysis is not immediate because by default all the packages involved in the communication are displayed. These include each level of the Bluetooth specification.

As previously seen, the management of services and characteristics is one of the tasks of the Generic Attribute Profile, which exploits the ATT protocol. Wireshark identifies it through the `CID 0x0004`. Therefore, it is possible to remove unwanted packages from the list and keep only the important ones by inserting the expression `btl2cap.cid == 0x0004` into the filter bar (the bar below the buttons on top of the window).

Once this is done, it becomes immediate to observe what are the characteristics on which a writing operation has been carried out, the written values and the content of the notifications received.

Let's take the previous image as a reference (see *Wireshark log example*). It shows an example of logging file generated by Android. The upper half of the image shows a list of all the exchanged packages (already filtered by ATT protocol) in time order. Each entry reveals the source, the destination and a brief description of the packet's content.



```
No.   Time        Source        Destination     Protocol Length Info
  124 14.179333   remote ()     localhost ()    ATT          15 Rcvd Find Information Response
  127 14.195621   localhost ()  remote ()       ATT          12 Sent Read Request, Handle: 0x0430
  128 14.209384   remote ()     localhost ()    ATT          12 Rcvd Read Response
  129 14.216888   localhost ()  remote ()       ATT          19 Sent Write Request, Handle: 0x0411
  132 14.298143   remote ()     localhost ()    ATT          10 Rcvd Write Response
  133 14.347386   remote ()     localhost ()    ATT          18 Rcvd Handle Value Notification, Handle: 0x0421
  134 31.961833   localhost ()  remote ()       ATT          14 Sent Write Request, Handle: 0x0411
  135 32.043119   remote ()     localhost ()    ATT          10 Rcvd Write Response
  136 32.141600   remote ()     localhost ()    ATT          18 Rcvd Handle Value Notification, Handle: 0x0421
  138 66.802989   localhost ()  remote ()       ATT          14 Sent Write Request, Handle: 0x0411
  139 66.899402   remote ()     localhost ()    ATT          10 Rcvd Write Response
  140 66.997462   remote ()     localhost ()    ATT          18 Rcvd Handle Value Notification, Handle: 0x0421

▶Frame 134: 14 bytes on wire (112 bits), 14 bytes captured (112 bits)
▶Bluetooth HCI H4
▶Bluetooth HCI ACL Packet
▼Bluetooth L2CAP Protocol
   Length: 5
   CID: Attribute Protocol (0x0004)
▼Bluetooth Attribute Protocol
   Opcode: Write Request (0x12)
   Handle: 0x0411
   Value: 4128

0000  02 40 00 09 00 05 00 04  00 12 11 04 41 28        .@...... ....A(
```

Fig. 13: Wireshark log example

By selecting a package and expanding the Bluetooth Attribute Protocol section (as in the lower half of the image), all the details appear:

- the type of operation performed (*read/write/notify*)
- the characteristic on which it was carried out (identified by a **16-bit Handle**)
- the transmitted data

The reason why the characteristic is identified by a **Handle** instead of the aforementioned UUID is that the ATT protocol considers characteristics, services and profiles as *attributes*. Each attribute is recognized through a handle.

Starting from the concepts described so far, the work of reverse engineering becomes applicable using various methodologies and considerations. In addition to identifying the written and read characteristics, the goal is to decipher *the meaning* of the transmitted data, which represent the proprietary communication protocol designed by the manufacturer. It is important to note that, depending on which is the target BLE device, there are operations that require data entry by the user to be performed (e.g. “set temperature to XX degrees”) and others that do not require external data (e.g. “turn off the valve “).

Operations requiring external data

These are operations for which the user must specify some details (i.e. *parameters*) so that they can be performed. In the programming world, this coincides with the invocation of a function that requires parameters: the function name remains the same, while the parameter(s) varies depending on the user's choice.

The value sent to the BLE device can therefore be divided into two sections:

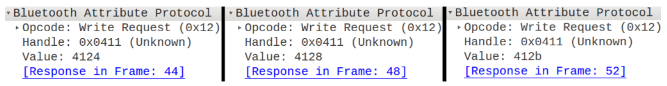
- a **common pattern**, representing the *operative code* of the instruction
- a **variable part** based on the information supplied by the user

NOTE: It is quite common that the variable part is somehow coded according to the choices of the producer instead of keeping "raw" values

The idea is to group in a single log file the execution/invocation of several operations that have the same operative code and for each of them to change the parameter. This technique makes it possible to identify the common pattern and, in the simplest cases, allows to understand the method used to calculate the variable part. In cases where it is not trivial to understand how the variable part has been encoded, the decompilation of the Android application becomes essential. This will be discussed later.

Example

This example refers to the radiator valves discussed in the *Introduction* and assumes to have already connected the BLE device to the central device and activated the packet sniffing.



```
▾Bluetooth Attribute Protocol
 ▸Opcode: Write Request (0x12)
  Handle: 0x0411 (Unknown)
  Value: 4124
  [Response in Frame: 44]
```

```
▾Bluetooth Attribute Protocol
 ▸Opcode: Write Request (0x12)
  Handle: 0x0411 (Unknown)
  Value: 4128
  [Response in Frame: 48]
```

```
▾Bluetooth Attribute Protocol
 ▸Opcode: Write Request (0x12)
  Handle: 0x0411 (Unknown)
  Value: 412b
  [Response in Frame: 52]
```

Fig. 14: Commands to set 18, 20 and 21.5 degrees

Following the idea described in the previous paragraph, we set the temperature consecutively at 18°C, 20°C and 21.5°C. By analyzing the log file via Wireshark (see *previous image*) is now possible to observe that the values were written on the handle `0x0411` and their values was respectively `0x4124`, `0x4128` and `0x412B`.

It is easy to notice that byte `0x41` appears in all three cases. It is therefore reasonable to suppose that it identifies the type of operation to be performed (i.e. it's the *operative code*). Consequently, the remaining byte will represent and encoding of the temperature we selected each time.

By converting the hexadecimal bytes `0x24`, `0x28` e `0x2B` to base 10, we obtain 36, 40 and 43. These correspond to twice the initial values (which were 18, 20, 21.5). It is therefore clear that the coding used consists in multiplying the desired temperature by two.

By using this information we deduced that:

- to send the "change temperature" command we have to write on the *handle* `0x0411`
- the value to be written is composed of `0x41` concatenated to twice the temperature we want to set

Operations not requiring external data

These are operations whose execution requires no external data. It's therefore reasonable to expect that the value sent to the valve to cause its activation consists only of an invariant operating code.

In these cases, the goal is to create log files that represent the execution of a single instruction, in order to identify immediately what has been transmitted to the BLE peripheral. It is important, however, to perform the same operation starting from different conditions in order to verify the effective invariability of the operating code in all the "states" in which the device can be. For this purpose, the decompilation of the Android application can provide support.

Example

This example refers to the radiator valves discussed in the *Introduction* and assumes to have already connected the BLE device to the central device.

Before starting the packet sniffing, we put the valve in *automatic mode*. Then we activate the packet sniffing and through the application we set the "boost mode". By analyzing the log file via Wireshark is now possible to observe that, once again, the command was sent to the handle `0x0411` and the written value is `0x4501`.

We repeat the same procedure twice more. The first time with the valve in "manual mode" and the second time in "holiday mode". By looking at the log files, we note that the values sent and the handle remain the same. This allows to conclude that the two bytes `0x4501` are the invariant operating code of the "start boost mode" instruction taken into account, regardless of the starting state in which the radiator valve can be (*automatic*, *manual* or *holiday* mode).

Notifications

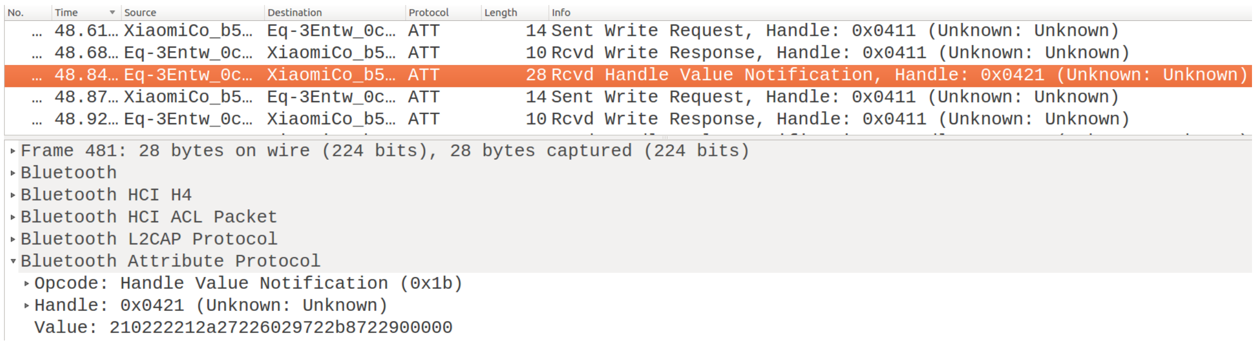
Notifications are sent from the BLE device to the central device following the execution of each command. It is not necessary to apply a specific method to detect them because they are already present in the log files created previously. Wireshark reports the presence of notifications through the description "Rcvd Handle Value Notification, Handle: `0xYYYY`" in the "Info" column (see *Wireshark log example*).

The structure (the carried value) of the notifications can be very variable and unpredictable, also because they could be sent to the central device at any time. They could contain a lot of information or indicate only a confirmation of correct execution. In general, as with *operations with parameters* , notifications are often composed of:

- **common patterns** which allow splitting notifications into groups/types
- **variable values** that provide detailed information coded according to criteria chosen by the manufacturer

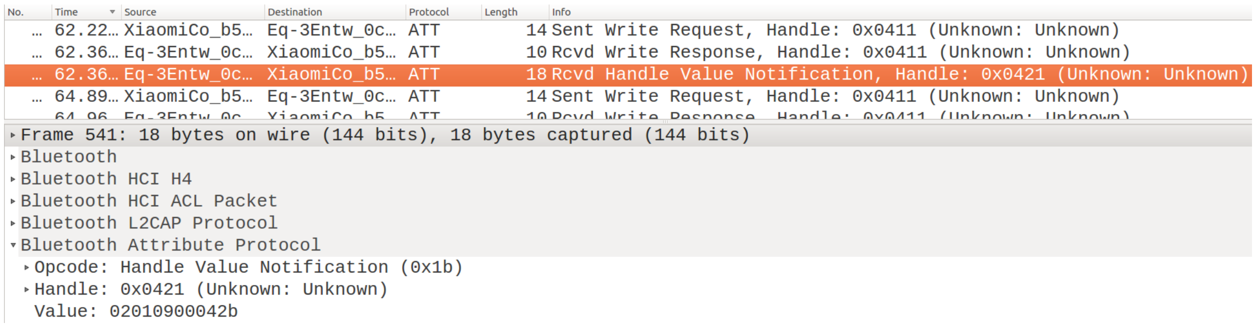
Example

This example refers to the radiator valves discussed in the *Introduction*.



```
No.  Time      Source          Destination     Protocol Length Info
…    48.61…   XiaomiCo_b5…   Eq-3Entw_0c…   ATT       14 Sent Write Request, Handle: 0x0411 (Unknown: Unknown)
…    48.68…   Eq-3Entw_0c…   XiaomiCo_b5…   ATT       10 Rcvd Write Response, Handle: 0x0411 (Unknown: Unknown)
…    48.84…   Eq-3Entw_0c…   XiaomiCo_b5…   ATT       28 Rcvd Handle Value Notification, Handle: 0x0421 (Unknown: Unknown)
…    48.87…   XiaomiCo_b5…   Eq-3Entw_0c…   ATT       14 Sent Write Request, Handle: 0x0411 (Unknown: Unknown)
…    48.92…   Eq-3Entw_0c…   XiaomiCo_b5…   ATT       10 Rcvd Write Response, Handle: 0x0411 (Unknown: Unknown)
▸Frame 481: 28 bytes on wire (224 bits), 28 bytes captured (224 bits)
▸Bluetooth
▸Bluetooth HCI H4
▸Bluetooth HCI ACL Packet
▸Bluetooth L2CAP Protocol
▾Bluetooth Attribute Protocol
  ▸Opcode: Handle Value Notification (0x1b)
  ▸Handle: 0x0421 (Unknown: Unknown)
   Value: 210222212a27226029722b8722900000
```

Fig. 15: Notif. after a “Daily Profile Request”



```
No.  Time      Source          Destination     Protocol Length Info
…    62.22…   XiaomiCo_b5…   Eq-3Entw_0c…   ATT       14 Sent Write Request, Handle: 0x0411 (Unknown: Unknown)
…    62.36…   Eq-3Entw_0c…   XiaomiCo_b5…   ATT       10 Rcvd Write Response, Handle: 0x0411 (Unknown: Unknown)
…    62.36…   Eq-3Entw_0c…   XiaomiCo_b5…   ATT       18 Rcvd Handle Value Notification, Handle: 0x0421 (Unknown: Unknown)
…    64.89…   XiaomiCo_b5…   Eq-3Entw_0c…   ATT       14 Sent Write Request, Handle: 0x0411 (Unknown: Unknown)
▸Frame 541: 18 bytes on wire (144 bits), 18 bytes captured (144 bits)
▸Bluetooth
▸Bluetooth HCI H4
▸Bluetooth HCI ACL Packet
▸Bluetooth L2CAP Protocol
▾Bluetooth Attribute Protocol
  ▸Opcode: Handle Value Notification (0x1b)
  ▸Handle: 0x0421 (Unknown: Unknown)
   Value: 02010900042b
```

Fig. 16: Notif. after “Manual Mode” command

We perform different types of operations and then look at the log files with Wireshark. By inspecting the package list, it is easy to see that the radiator valve always sends at least one notification at the end of each operation. Notifications are identified by packages called “Rcvd Handle Value Notification, Handle: 0x0421” (see *Notif. after a “Daily Profile Request”* and *Notif. after “Manual Mode” command*). The values contained will often be very different, both in terms of content and length (number of byte sent).

The process followed to understand the meaning of the received values is equivalent to that reported in the section *Operations requiring external data*. It is therefore necessary to group the notifications received as a result of the same command in the same log file and observe the differences.

The result of this activity highlights a large subdivision carried out by the *most significant bytes*: `0x0202[..]` and `0x21[..]` indicate notifications relating to the writing and the request of a daily profile, while `0x0201[..]` identifies those resulting from the execution of any other operation. Even in this case the decompilation of the Android application becomes useful to better understand the syntax. This will be discussed in the next section.

NOTE: the value of the handle is always the same and identifies the characteristic on which notifications are sent.

Data Sent Through Advertising Packets

As already mentioned in section *BT/BLE: Main Differences*, the Bluetooth Low Energy standard allows an exchange of data to be performed *without a pairing procedure*. Basically, in these cases, the BLE device sends data in broadcast to central devices without using notifications. This is done by taking advantage of the **Advertising packages**.

The structure of this type of package is described in the Bluetooth specifications[17]. Being another protocol, it is different (and more complex) from the structure of the ATT packages seen previously: the *OpCode*, *Handle* and *Value* fields are no longer present.

Note: We have *previously shown* how to filter in Wireshark only the packets related to the ATT protocol. However, the Advertising packages are related to the HCI protocol. You can therefore keep only these packages by writing `bthci_evt` into the filter bar (the bar below the buttons on top of the window).

We do not need to know the structure of the advertising packages in detail, but it is important to know that:

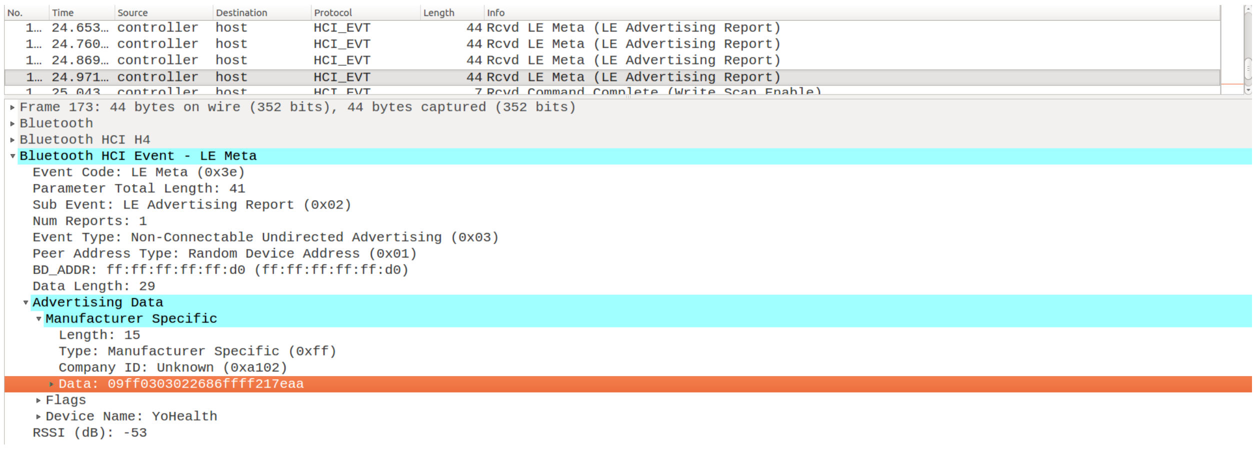
- a **Address** field contains the address of the device that is doing advertising
- **a Data field contains the data we are interested in, formatted as:**
  - `length`: number of bytes of *AD type* + *AD data*
  - `AD type`: Identifies the type of data present in *AD Data*. The possible data types and their related meanings are defined in the "Bluetooth Core Specification Supplement"[18]
  - `AD data`: Payload. Its length depends on the *AD Type* field

Wireshark identifies advertising packages through the description "Rcvd LE Meta (LE Advertising Report)" in the "Info" column. The *previous image* shows the presence of all the described fields, albeit with slightly *different names*.

The *Address* field is called `BD_ADDR`, while the *Data* field we are interested in is the one shown under the `Advertising Data > Manufacturer Specific` section. According to the specifications[18], "Manufacturer Specific" is an *AD type* represented by the code `0xFF` and must be at least 2 Bytes long.

[17] Bluetooth Core Specification 5.0, Volume 2, Part E, Page 1193

[18] Bluetooth Core Specification Supplement, Part A



```
No.  Time      Source      Destination  Protocol  Length  Info
1…   24.653…   controller  host         HCI_EVT       44  Rcvd LE Meta (LE Advertising Report)
1…   24.760…   controller  host         HCI_EVT       44  Rcvd LE Meta (LE Advertising Report)
1…   24.869…   controller  host         HCI_EVT       44  Rcvd LE Meta (LE Advertising Report)
1…   24.971…   controller  host         HCI_EVT       44  Rcvd LE Meta (LE Advertising Report)
1    25.043    controller  host         HCI_EVT        7  Rcvd Command Complete (Write Scan Enable)
▸ Frame 173: 44 bytes on wire (352 bits), 44 bytes captured (352 bits)
▸ Bluetooth
▸ Bluetooth HCI H4
▾ Bluetooth HCI Event - LE Meta
    Event Code: LE Meta (0x3e)
    Parameter Total Length: 41
    Sub Event: LE Advertising Report (0x02)
    Num Reports: 1
    Event Type: Non-Connectable Undirected Advertising (0x03)
    Peer Address Type: Random Device Address (0x01)
    BD_ADDR: ff:ff:ff:ff:ff:d0 (ff:ff:ff:ff:ff:d0)
    Data Length: 29
  ▾ Advertising Data
    ▾ Manufacturer Specific
        Length: 15
        Type: Manufacturer Specific (0xff)
        Company ID: Unknown (0xa102)
      ▸ Data: 09ff0303022686ffff217eaa
    ▸ Flags
    ▸ Device Name: YoHealth
    RSSI (dB): -53
```

Fig. 17: Wireshark log of Advertisement packets

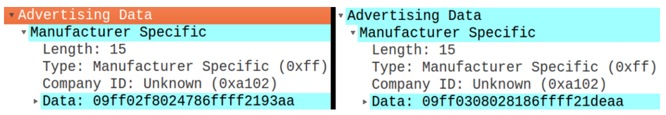
As for *notifications*, the contents (the carried value) of advertising packets can be very variable. Also in this case, the `Data` field will probably be composed of a **common pattern** and a **variable part**. The latter contains the information we are looking for, coded according to some criteria chosen by the manufacturer.

The idea is always the same: to generate *more log files* and, for each one, to provide different input to the advertising device. For example, if the device is a BLE scale, use objects/people with different weights. Then look at the changes in the `Data` field. These changes are related to the inputs I have provided to the device and therefore are useful to understand their meaning.

Example

This example refers to the BLE scale discussed in the *Laica PS7200L Protocol* section.

Activate Bluetooth and packet sniffing on your Android device. Weigh a person using the application supplied with the scale. Once the log file is obtained, repeat the same procedure with another person or an object of different weight.



```
▾ Advertising Data                          ▾ Advertising Data
  ▾ Manufacturer Specific                     ▾ Manufacturer Specific
      Length: 15                                  Length: 15
      Type: Manufacturer Specific (0xff)          Type: Manufacturer Specific (0xff)
      Company ID: Unknown (0xa102)                Company ID: Unknown (0xa102)
    ▸ Data: 09ff02f8024786ffff2193aa            ▸ Data: 09ff0308028186ffff21deaa
```

Fig. 18: Log with two different weights (*76kg left*, *77.6kg right*)

We look at the values contained in the `Advertising Data > Manufacturer Specific > Data` field in the Advertising packages (see *image above*). We compare them:

```
09 ff 02 f8 02 47 86 ff ff 21 93 aa
09 ff 03 08 02 81 86 ff ff 21 de aa
```

A good part of the data transmitted remains **constant** and only a few bytes change (the *second*, the *third* and the *second to last*). We have thus understood where to look for the data that interests us (in the **variable part**).

For instance, in this case it is rather easy to notice that the *first two bytes* (of the variable part) concatenated return the weight of the person multiplied by 10, as:

- `02 F8` in decimal base corresponds to `760` (*76.0 if divided by 10*)
- `03 08` in decimal base corresponds to `776` (*77.6 if divided by 10*)

In conclusion, applying what has just been described to all the functionalities of a BLE device allows to identify a good part of the communication protocol. Probably some details related to the variable data within the various commands remain hidden. The next section discusses how to fill these gaps and confirm what has been deduced through the analysis of the application code.

2.2.4 Android Application Analysis

This section wants to give a guideline and provide some reference examples about: * which services to use to decompile an Android application * how to analyze the aspects of communication and data transmission via Bluetooth

The fact that there is no common standard to follow in the creation of Android applications makes it difficult to generalize the code and the analysis.

In general, to get the source code you need the **APK package** of the application, which is basically an archive that contains all the data that Android needs to manage the installation of the app. Once in possession of the APK file, there are decompilers (typically for Java code, on which Android is based) that in a short time generate an archive containing the original code. The product code is automatically identified as “Android project” by Android Studio, which is the official open source *IDE* for the development of Android applications. This ensures ease of reading and analysis, taking advantage of the advanced development features provided by the software.

Get The APK Package

Option 1: through a web service

A fairly simple way to perform this step is to rely on a web service to extract from the Google Play Store the APK package of the application used by the BLE device. The APKPure site is the only one on which we have found our reference application, but there are also other sites that provide the same service, such as App Apk and APKSFree.

If none of these sites leads to a result, it is often possible to obtain the APK package through the procedures described in the next section.

Option 2: through a File Manager app

Also the *File Manager* applications often allow to extract the APK packages from the applications on the device. We mention this option as “second possibility” because in the medium case these applications require a lot of permissions to operate (e.g. read storage, read phone status). We report the procedure to be performed with two different applications:

ES File Manager (release 4.1.8.1)

1. from the application's main page (called "Home"), select *"APP"*
2. identify the application for which you want to get the apk
3. press and hold the icon, then click on "backup"
4. the app tells you where the apk file is saved (in our case `/sdcard/backups/apps`)

Astro File Manager (release 6.4.0)

1. from the application's main page (called "File Manager"), select *"Go To App Manager"*
2. identify your application, hold the icon and select "backup" from the drop-down menu
3. an apk file is created and it can be accessed from the file manager itself (it does not say where it is saved; in our case in `/sdcard/backups/apps`)

As you can see, the procedures are very similar and also on other applications will not be very different. Note that probably the well-known Titanium Backup allows you to do this. We did not consider it for simplicity, as it requires the device to be rooted.

Option 3: through ADB (manual method)

Assuming you have the application installed on your smartphone, you can get the APK package through **adb** (*Android Debug Bridge*), which is a command-line tool that allows you to control an Android smartphone via USB. The adb tool can be installed through the `android-adb-tools` package, available in the Debian repositories.

Note: `adb` requires USB debugging to be enabled. This function can be found in the "Developer Options" section. (refer to *this note* if there is no "Developer Options" entry in the settings of your smartphone)

Once installed adb and connected the device via USB, use the following commands:

```
$ adb shell pm list packages          #find the package name of the app
$ adb shell pm path package-name      #find the address of the app
$ adb pull app-address                #copy the apk
```

Depending on the Android version used, the following error may be returned:

```
remote object "app-address" does not exists
```

In this case, you need to move the apk file to an accessible folder before downloading it. Use the following commands:

```
$ adb shell cp app-address /storage/emulated/0/Download
$ adb pull /storage/emulated/0/Download/base.apk
```

The apk file `base.apk` should now be in your home and contains the necessary to get the Java code.

Obtain the source code

Using the APK file we now want to extract the Java code that makes up the application. Before proceeding it is good to know that there are cases in which the code will be obfuscated. *Code obfuscation* is a practice that consists in making the code more complicated without changing its functionality in order to make it more difficult to understand. This is done to avoid reverse engineering practices.

There are some open source Java de-obfuscators, such as Java Deobfuscator (and his GUI) or Enigma. They require a *Jar* file as an input, which can be obtained from the APK by following the first two points of *this* section. However, their use will not be covered in this guide. **Option 1: through a web service**

As before, we can delegate the work to web services and more than one site can be useful.

Among the sites tested, Java Decompilers is the one that provided the clearest code: probably the most similar to the real one (variable names are reasonable and there are no `GOTO` statements). ApkDecompilers produces the same result, but puts all the files in a single directory. Since Android applications include a lot of files, it is less comfortable.

Option 2: through dex2jar and JD-GUI (manual method)

Dex2jar and *JD-GUI* are two programs that respectively allow to transform the apk file into a *Jar* (Java Archive) and decompile it by following these steps:

1. Download the latest release of *dex2jar* from this page

   The file to be downloaded is called `dex-tools-X.X.zip`, where X.X indicates the version number (2.0, at the time of writing)

2. Execute these commands in a terminal:

   ```
   $ unzip dex-tools-X.X.zip
   $ cd dex2jar-X.X
   $ chmod u+x *.sh
   $ ./d2j-dex2jar.sh /path/to/application.apk     #the application␣
   ↪APK
   ```

   This produces a jar file in the `dex-tools-X.X` directory.

3. Download and install JD-GUI (released under the GPLv3 license)

4. Open *JD-GUI* and select `File > Open File` in order to open the jar archive produced with dex2jar.
5. Select `File > Save All Sources`. This produces a zip archive containing all the Java files.

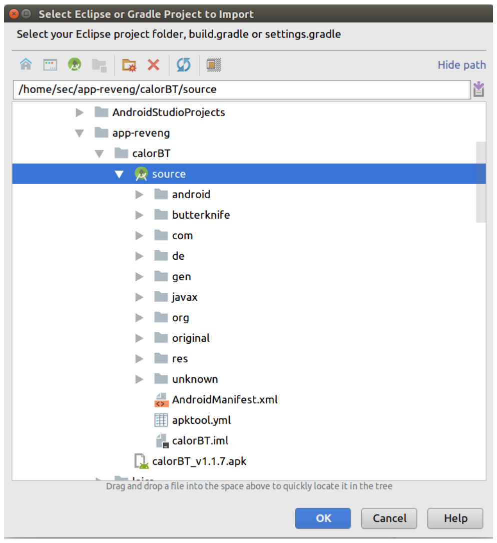
JD-GUI has the advantage of clearly showing the *Java packages* that make up the project. However, individual files are less clear than those produced with the service discussed in the *Option 1*.

Other decompilers can be used instead of JD-GUI. One of this is CFR, which is *not open source* but it also decompiles modern **Java 9 features**. Other good solutions are Fernflower and the one included in the Procyon suite.

Import In Android Studio

Regardless of the method chosen, the files obtained can be imported into Android Studio to be analyzed. Both *Option 1* and *Option 2* produce a zip file.

Note: Obviously, you can use any text editor to analyze the produced files. This guide focuses on Android Studio as it's an *open source* tool and now a standard in creating Android applications. Whatever the choice, we strongly suggest the use of an IDE that integrates well with Java. (e.g Eclipse, NetBeans).


Select Eclipse or Gradle Project to Import

Select your Eclipse project folder, build.gradle or settings.gradle

/home/sec/app-reveng/calorBT/source

Drag and drop a file into the space above to quickly locate it in the tree

Fig. 19: Android Studio - Import Project

Once extracted the zip file, open Android Studio and follow these steps:

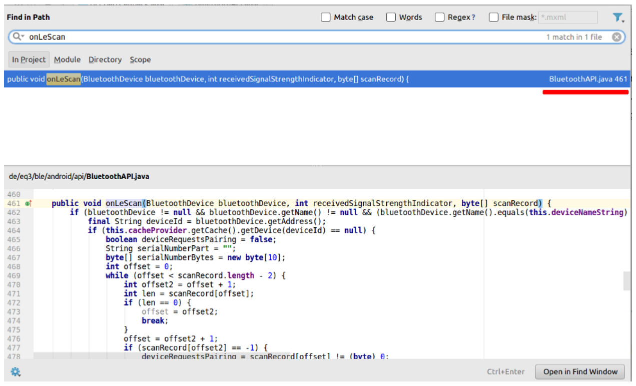
1. `File > New > Import Project`
2. Select the directory in which the archive was extracted (the one in which the *XML Manifest* file is present) and click `Next` (see *Android Studio - Import Project*).

3. `Create project from existing sources > Next`
4. Choose a name for the project and the directory in which to save it (personal choices). Then click `Next`.
5. All the default settings should be fine, so keep clicking `Next`.

After that, a panel on the left allows you to navigate between packages and files, while the right side shows an editor enabled for grammatical (and syntactic) correction.

What to look for in the code

In general, a good way to avoid having to consult all the files in the project is to proceed by keywords and use a search tool. In android studio this tool is provided in the `Edit > Find > Find In Path` menu. For example, the *next* image shows all the files containing the search keyword *OnLeScan*.



onLeScan

1 match in 1 file

public void onLeScan(BluetoothDevice bluetoothDevice, int receivedSignalStrengthIndicator, byte[] scanRecord) { BluetoothAPI.java 461

de/eq3/ble/android/api/**BluetoothAPI.java**

```java

    public void onLeScan(BluetoothDevice bluetoothDevice, int receivedSignalStrengthIndicator, byte[] scanRecord) {
        if (bluetoothDevice != null && bluetoothDevice.getName() != null && (bluetoothDevice.getName().equals(this.deviceNameString)
            final String deviceId = bluetoothDevice.getAddress();
            if (this.cacheProvider.getCache().getDevice(deviceId) == null) {
                boolean deviceRequestsPairing = false;
                String serialNumberPart = "";
                byte[] serialNumberBytes = new byte[10];
                int offset = 0;
                while (offset < scanRecord.length - 2) {
                    int offset2 = offset + 1;
                    int len = scanRecord[offset];
                    if (len == 0) {
                        offset = offset2;
                        break;
                    }
                    offset = offset2 + 1;
                    if (scanRecord[offset2] == -1) {
                        deviceRequestsPairing = scanRecord[offset] != (byte) 0;
```

Fig. 20: Android Studio “Find in path” window

Within the reverse engineering activity there is little interest in understanding how the connection to the target BLE device works. We can therefore ignore the details of the bluetooth connection to focus on two things:

1. identify the files that create the **strings that will be sent to the peripheral** (made of hexadecimal values), to see how they are composed
2. identify the files that manage **received notifications** to understand how they are interpreted

If you want to find Java classes that deals with the act of Bluetooth connection, you can identify useful keywords (such as `BluetoothAdapter`, `startLeScan`, `LeScanCallback`, `BluetoothGatt`) by referring to the Android BLE API page.

Commands

Finding the classes and methods that deal with composing commands (i.e. *strings* of values) is not straightforward. This is because every manufacturer can act without constraints when he implements his own proprietary coding. It is not even necessary for these classes to contain keywords related to Bluetooth, as they could be treated separately.

In the best case, the application will contain Java packages with names like *command*, *sendCommand*, or something similar and this facilitates the search. Otherwise, the criterion to be used, consists of:

- to use as search keywords the name of the methods that Android uses to write BLE characteristcs
- starting from the results found, **go back** to the classes used for the generation of commands

Specifically, Android within the BluetoothGatt class provides the `writeCharacteristic` method. Note that the values we are interested in are those that must be written on the characteristics. To set a value before writing it, the BluetoothGattCharacteristic class provides the `setValue` method.

As a result, `command`, `writeCharacteristic` and `BluetoothGattCharacteristic` are examples of good search keys.

Notifications

Notification management can also be done in various ways depending on the choices of the producers. However, as for commands, at some point the application must definitely use the default methods provided by Android to manage the reception.

Hypothetically, once received and extracted the content of the notification, this is sent to a sort of *parser* that interprets it and then performs other tasks depending on its meaning. This parser is what we are interested in.

To easily identify files that contain useful code, it is important to know that:

- the receipt of notifications by a feature must be **explicitly enabled**, using the `setCharacteristicNotification` method of the BluetoothGatt class
- the callback function called when a notification is received is `onCharacteristicChanged`, provided by the BluetoothGattCallback class

This callback function will be the starting point to go back to the pieces of code that actually interpret the content of the notifications.

In conclusion, good search keys are: `setCharacteristicNotification`, `onCharacteristicChanged` and `BluetoothGattCallback`.

Create A Class Diagram

It is often useful to have a Class Diagram[6] available for the whole project or just for small parts. This allows you to schematically represent the application and **highlight the dependencies** between the classes that compose it.

[6] Wikipedia - Class Diagram

Manually creating a Class Diagram for an Android application can result in a lot of work, so some automated tools will help us. There are several Class Diagram generators for Java/Android code, but we will explain how to install and use Code Iris because:

- integrates easily into Android Studio
- allows to *filter* and *highlight* classes and packages (useful for big projects)
- it is quite frequently updated
- allows to export data in Json format

Note: *Code Iris is not open source*. However, after trying different free alternatives, our opinion is that it is the easiest and most complete solution. If you want to stick with free software, skip to *this* paragraph.

A few steps are required to install *Code Iris*:

1. open Android Studio and select `File > Settings > Plugins`
2. click on `Browse Repositories` on the bottom of the window
3. search for `code iris` and click on the green "”Install”" button
4. restart Android Studio

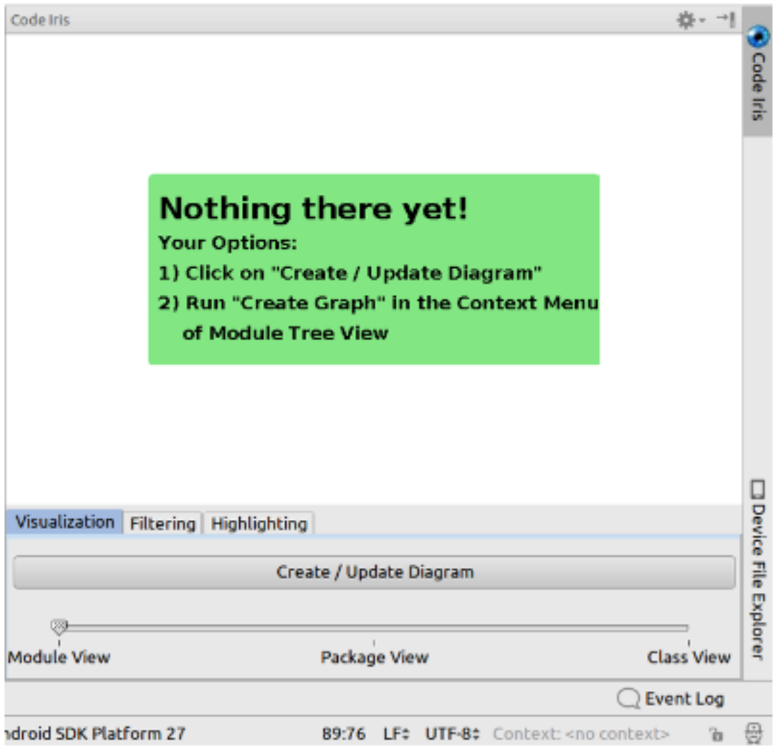

Nothing there yet!
Your Options:
1) Click on "Create / Update Diagram"
2) Run "Create Graph" in the Context Menu of Module Tree View

Fig. 21: Code Iris First Start

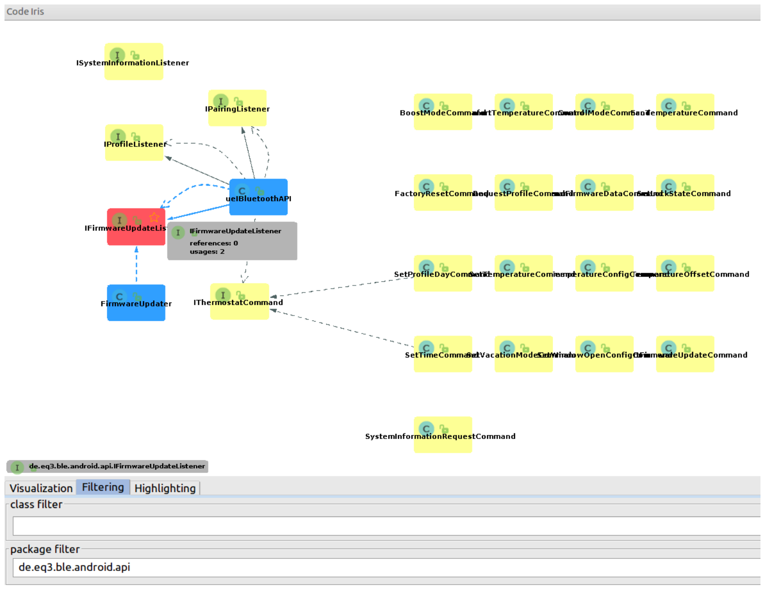
Once you restart Android Studio and open a project, you can start *Code Iris* through the corresponding tab at the top right of the window. The first time you need to create the Class Diagram via the "Create/Update Diagram" button (see *Code Iris First Start*).

The operation takes a few moments, but produces a Class Diagram related to the *whole project*. The result can be inspected via three different "views":

- **Module View**: the most abstract view, generally not very useful for our purpose
- **Package View**: maintains the subdivision into packages and also shows the classes they contain
- **Class View**: shows all classes without the subdivision into packages

The *Package View* and the *Class View* are both useful, depending on personal needs. The large size of Android projects, however, requires the ability to "cut" parts of the Class Diagram, so that it becomes easy to read.

Code Iris provides filtering tools according to **class name** and/or **package name**. This allows us to identify useful parts of the Class Diagram. Once this is done, moving the cursor on the names of the individual classes highlights all their dependencies (see *Class View with Filtering Enabled*).



class filter:

package filter: de.eq3.ble.android.api

Fig. 22: Class View with Filtering Enabled

NOTE: the version of Code Iris in the Android Studio repositories is from 2014. More recent versions (updated to 2018) can be easily installed by downloading the package from the JetBrains site and through the "Install Plugin From Disk" option in `File > Settings > Plugins` (inside Android Studio).

A free and open source alternative

Among the open source alternatives there are many software that can generate class diagrams[8][9][10][11]. However, they often present problems such as:

- Request to *manually* add each class to the diagram. It does not make much sense because we don't know how the application is composed. It's more important to be able to add individual packages.
- Request to *manually* select each Java source directory. Not ideal, given the size of Android projects.
- Very confusing graphic interface.

An acceptable solution involves the use of NetBeans IDE, which is a Java IDE[7] , and the EasyUML plugin. Since *NetBeans* is already present in the Debian repositories, you can install it with *apt*:

```
$ sudo apt install netbeans
```

EasyUML requires you to download the correct package, which depends on the NetBeans version, from this page. To install it you need to extract the files from the downloaded *zip* archive, open *NetBeans*, select `Tools > Plugins > Downloaded > Add Plugins` and choose all the `.nbm` files from the extracted folder.

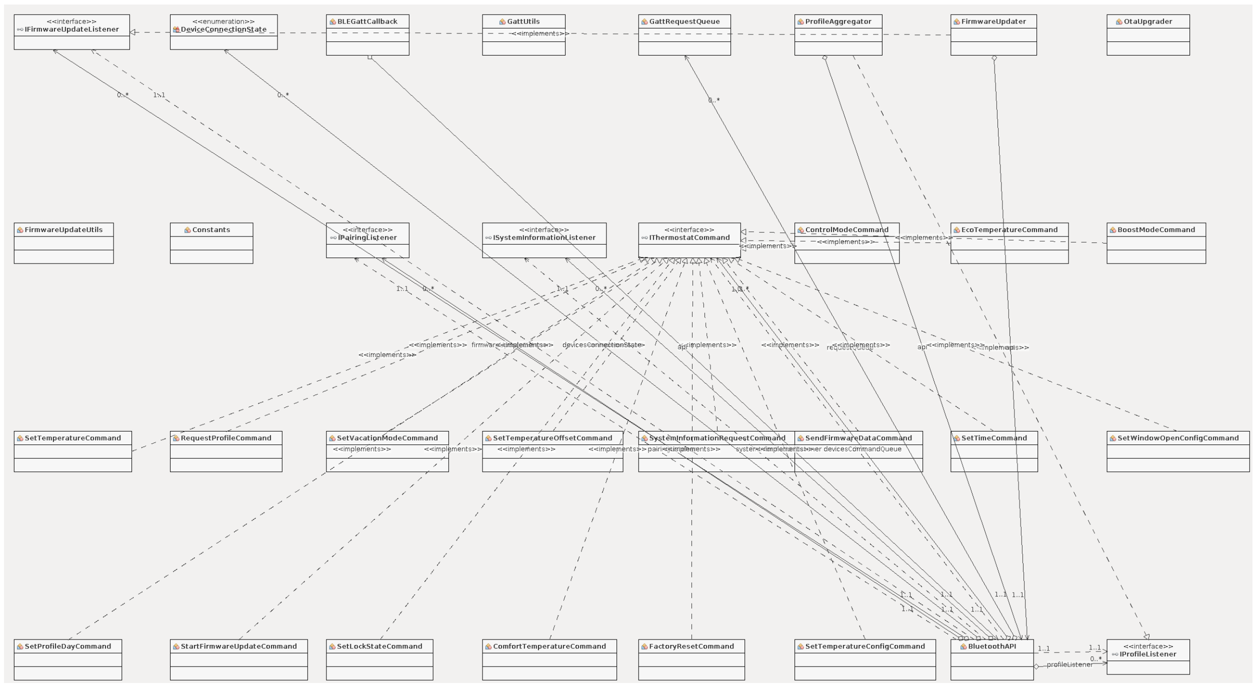
Once the software is working, both a Java project and a UML project must be created in order to obtain a Class Diagram. The following is a step by step guide:

1. **Create a Java project from existing sources:**
    - `New > Project > Java > Java project with existing sources > Next`
    - set the name of the project and when asked for "Source Package Folders" select the directory in which the APK' source code have been extracted
    - if the message "The specified package folder contains compiled class files" appears, click on *Ignore*
2. **Create an UML project:**
    - `New > Project > UML > UML Diagrams Project > Next`
3. Now the left panel shows the Java *packages*. Right click on the package you want to create the Class Diagram. and choose "easyUML Create Class Diagram".
4. Choose the easyUML project created before and click on "Create Class Diagram"

The generated Class Diagram contains *by default* more information (methods and members) than the one shown in the previous section. This makes it more detailed, but messy and often unclear.

[8] ArgoUml
[9] Umbrello UML Modeller
[10] BOUML
[11] Modelio
[7] Wikipedia - Integrated Development Environment

To keep only the names of the classes (and their relationships) make a right click on the diagram and remove the check from "Show Members" under "Visual Options". The following image shows the obtained diagram. For comparison purposes, the code part shown in the diagram is equivalent to the one shown in figure *Class View with Filtering Enabled*.



<<interface>> IFirmwareUpdateListener
<<enumeration>> DeviceConnectionState
BLEGattCallback
GattUtils
GattRequestQueue
ProfileAggregator
FirmwareUpdater
OtaUpgrader
FirmwareUpdateUtils
Constants
<<interface>> IPairingListener
<<interface>> ISystemInformationListener
<<interface>> IThermostatCommand
ControlModeCommand
EcoTemperatureCommand
BoostModeCommand
SetTemperatureCommand
RequestProfileCommand
SetVacationModeCommand
SetTemperatureOffsetCommand
SystemInformationRequestCommand
SendFirmwareDataCommand
SetTimeCommand
SetWindowOpenConfigCommand
SetProfileDayCommand
StartFirmwareUpdateCommand
SetLockStateCommand
ComfortTemperatureCommand
FactoryResetCommand
SetTemperatureConfigCommand
BluetoothAPI
<<interface>> IProfileListener
<<implements>>
profileListener

Fig. 23: Netbeans And EasyUML - Class Diagram

Example Application Analysis

The examples shown will temporarily refer to the *CalorBT* application, supplied with the radiator valves discussed in the *introduction*. They will be subsequently extended to other devices as soon as they are tested.

The *CalorBT* application, developed by eQ-3, is available for Android 4.4+ and iOS 8.3+ platforms. The analyzed version is the 1.1.7 updated to the month of January 2016 (currently the last available), for which the source code was obtained using the *APKPure* and *JavaDecompilers* web services, as described in the previous sections (see *Get The APK Package* and *Obtain the source code*).

The application is quite simple and the initialization phase only requires to pair the central device with the desired radiator valves through a simple pairing procedure. It is interesting to note that it's allowed to communicate with only **one device at a time**. Once the connection is made, you are sent to a temperature management activity (see *Main Activity*), from which you can access all the other features.

Once the project has been imported into Android Studio, it's immediately obvious that the source

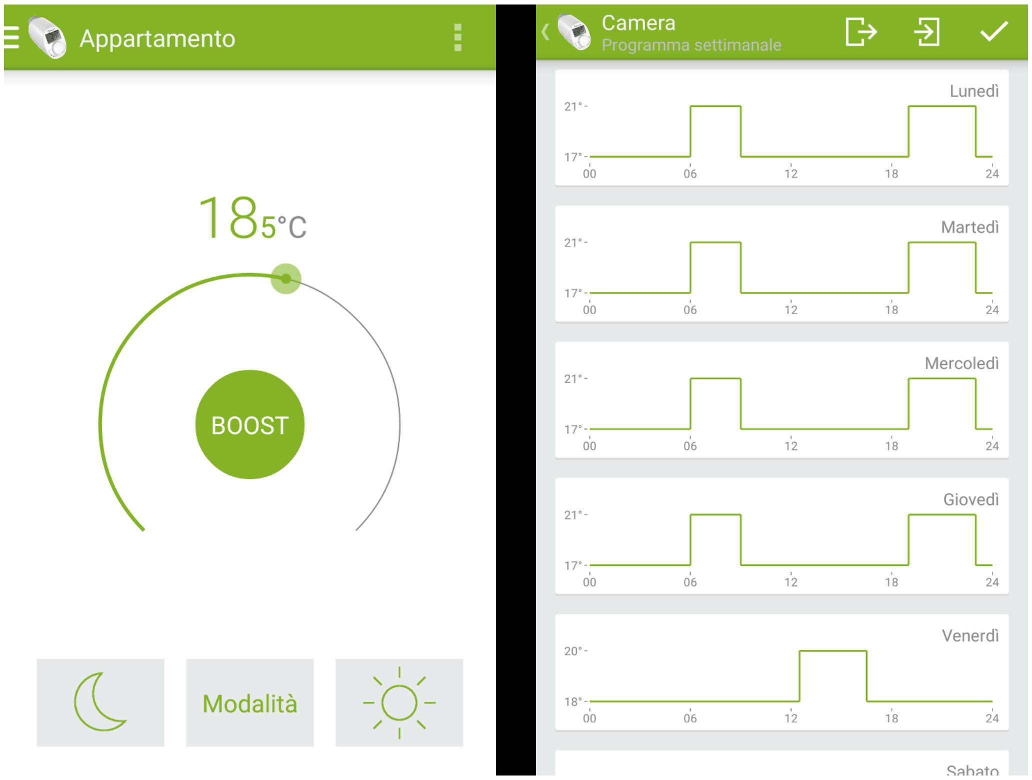


Fig. 24: Main Activity (left) and Weekly Schedule (right)

code is *free of obfuscation*. The files are completely readable and mutually consistent. Furthermore, the decompiler maintained the package subdivision of the entire application.

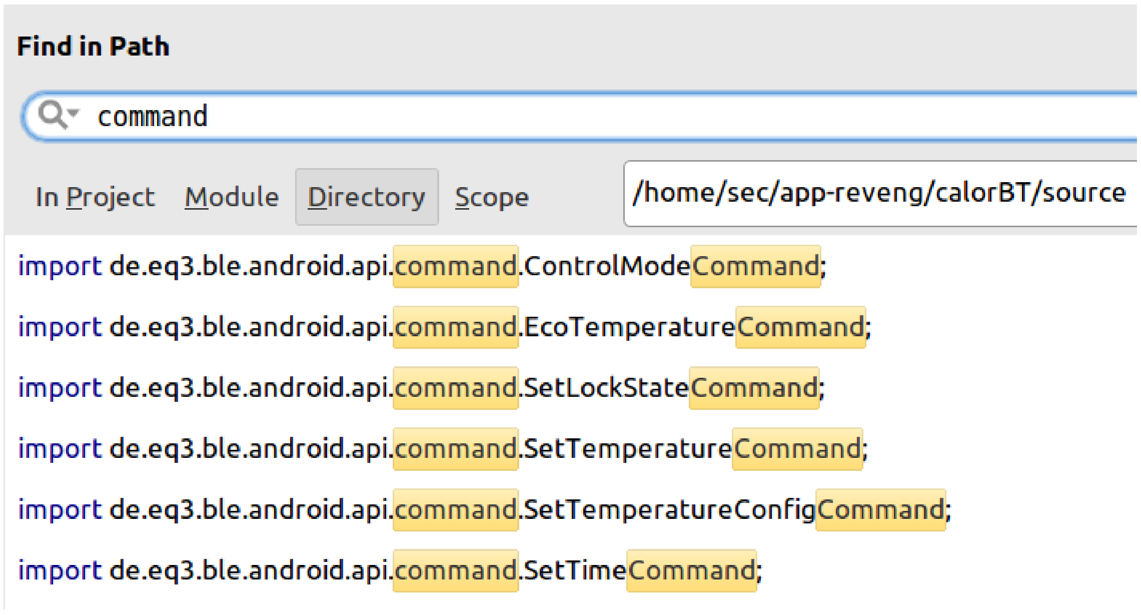
This is good news, it suggests that the manufacturer has not tried to prevent reverse engineering works. Nevertheless, the analysis is not immediate: excluding external libraries (such as *ButterKnife* and *FasterXML*) the application consists of 1398 files including **144 classes** and **16 Java interfaces**. Given such a large number of files, it will be necessary to use the search methods discussed in the previous sections.

Commands

Proceeding as described in the previous section, a short search with the keyword `command` shows the existence of a series of Java classes that form the code needed to compose the commands (see *Search results*). They are contained in the `de.eq3.ble.android.api.command` package (*a Java class for each command*).

This type of organization is particularly helpful as it allows to know all the instructions that the application can send to the radiator valve and for each one provides detailed information. Each class has been designed according to the same principle:

- the constructor deals with composing and storing the value corresponding to the instruction in an array
- external activities can request this value through the public method `getCommandData`,



```
import de.eq3.ble.android.api.command.ControlModeCommand;
import de.eq3.ble.android.api.command.EcoTemperatureCommand;
import de.eq3.ble.android.api.command.SetLockStateCommand;
import de.eq3.ble.android.api.command.SetTemperatureCommand;
import de.eq3.ble.android.api.command.SetTemperatureConfigCommand;
import de.eq3.ble.android.api.command.SetTimeCommand;
```

Fig. 25: Search results

which is certainly present as required by the `IThermostatCommand` interface implemented by each class.

The example seen in the "*Operations requiring external data*" section refers to the `SetTemperatureCommand` class shown below.

```java
public class SetTemperatureCommand implements IThermostatCommand {

    private final byte[] commandData;

    public SetTemperatureCommand(Number setPointTemperature) {
        this.commandData = new byte[2];
        this.commandData[0] = (byte) 65;
        this.commandData[1] = (byte) ((int)
            (setPointTemperature.doubleValue() * 2.0d));
    }

    public byte[] getCommandData() {
        return this.commandData;
    }
}
```

In summary, the logging activity had identified:

- the byte `0x41` as a common pattern to all instructions of this type
- a second byte as a variable part, equivalent to the chosen temperature *doubled*

Line 6 of the above code shows that the command will consist of 2 bytes, the content of which is shown in lines 7 and 8. The first always contains the value 65, whose conversion in hexadecimal corresponds precisely to `0x41`. The second one, on the other hand, is variable. It's based on the `setPointTemperature` parameter supplied to the class constructor and reveals the temperature coding already mentioned.

It is interesting to note that line 8 not only provides information on the meaning of the second byte, but also indicates **how to calculate** it. This is particularly helpful in creating the instructions within the management software discussed later, so that they can be consistent with what is required by the valve.

Notifications

Referring to what was said in the *previous section*, we look for the function `onCharacteristicChanged`. This is located in the file `BLEGattCallback.java`. Looking at the code, we see that the content of the notification is interpreted by the `updateDeviceState` function, whose prototype is:

```
public void updateDeviceState(String deviceId, byte[] value)
```

The function is located in the file `BluetoothAPI.java` and, as for commands, the data are stored in a *byte array* (represented by the second argument `value`).

`updateDeviceState` recognizes the type of notification by reading the value contained in the first and, in cases of ambiguity, in the second byte. Based on this, it delegates the correct operations to other methods. This creates a first subdivision, corresponding to the one mentioned in the *logging section*. More detailed information is then obtained by analyzing function calls.

The method adopted is therefore very simple. Assuming, for example, to send a request to read a daily profile to the radiator valve, the result will consist of a notification whose first byte will correspond to the value `0x21` (i.e. 33, in the decimal system). Within the `updateDeviceState` function, which will not be reported entirely for space reasons, this is the part of code involved:

```
1  int frameType = value[0] & 255; //extract the first byte
2  ..
3  if (frameType == 33) {
4
5      dayOfWeek = ModelUtil.getDayOfWeek(value[1]);
6      byte[] profileData = new byte[(value.length - 2)];
7
8      for (int i = 0; i < profileData.length; i++){
9          profileData[i] = value[i + 2];
10     }
11
12     this.profileListener.profileDataReceived(dayOfWeek, profileData);
13 }
14 ..
```

As shown in line 12, the management of received data is delegated to the `profileDataReceived` function. It uses two parameters:

- the day of the week, coded according to the static method `getDayOfTheWeek` (line 5)
- the `profileData` array, which essentially corresponds to the received notification value, *excluding* the two most significant bytes (that have already been used)

Basically, it is a matter of analyzing the two functions mentioned above, which allow to examine in depth the aspects related to the semantics of each byte. The first one, allows us to understand how the days of the week are coded within the application: `0` means *Saturday*, `1` means *Sunday* and so on.

```
public static DayOfWeek getDayOfWeek(byte b) {
    switch (b) {
        case (byte) 0:
            return DayOfWeek.SATURDAY;
        case (byte) 1:
            return DayOfWeek.SUNDAY;
        case (byte) 2:
            return DayOfWeek.MONDAY;
        case (byte) 3:
            return DayOfWeek.TUESDAY;
        case (byte) 4:
            return DayOfWeek.WEDNESDAY;
        case (byte) 5:
            return DayOfWeek.THURSDAY;
        default:
            return DayOfWeek.FRIDAY;
    }
}
```

The second one shows how to interpret all the remaining bytes. `Here` is the original code extracted from the application. We report the main points to understand how it works.

```
public void profileDataReceived(DayOfWeek dayOfWeek, byte[]␣
↪profileData) {

    //creates a list of pairs (temperature, time)
    List<ProfileDataPair> dataPairs = new ArrayList();

    for (i = 0; i < profileData.length; i += 2) {
        //reads two byte at a time and creates (temperature, time)␣
↪pairs
        int time = (profileData[i + 1] & 255) * 10;
        dataPairs.add(new ProfileDataPair(((double) profileData[i]) /␣
↪2.0d, time));
        if (time == 1440) break;
    }
```

(continues on next page)

(continued from previous page)

```
    //find the base temperature to keep outside the programmed ranges
    double baseTemperature = getBaseTemperature(dataPairs);
    ...

    //create a list of Period.
    //each Period contains the data of a range programmed by the user
    List<Period> periods = new ArrayList();

    for (i = 0; i < dataPairs.size(); i++) {

        ProfileDataPair pair = (ProfileDataPair) dataPairs.get(i); //
↪get a Pair

        //if the temperature of the pair is different from the base␣
↪temperature
        //then the user has entered a schedule for a certain period
        if (pair.temperature != baseTemperature) {

            Period currentPeriod = new Period(); //create a Period

            if(i>0){
                //the start time of the Period is the end of the␣
↪previous pair
                currentPeriod.
↪setStarttimeAsMinutesOfDay(((ProfileDataPair) dataPairs.get(i - 1)).
↪time);
            }
            ...

            //no more than 3 periods can be set
            if (periods.size() < 3)
                periods.add(currentPeriod);
        }
    }

    ...
    //show the interpreted data in the application
}
```

We can therefore deduce that:

- The bytes of the received notification must be interpreted as consecutive pairs `(temperature, time)`. Each pair indicates the temperature to be kept until a certain time.
- In each pair the time is multiplied by 10, while the temperature is divided by 2

- It is possible to identify the base temperature with the algorithm described by the function `getBaseTemperature`
- The ranges programmed by the user are those that don't use the base temperature and are *at most three*.

This allows us to understand the meaning of the entire notification. An example of the possible values received is shown in the *Notifications* section dedicated to the valve protocol.

Finally, it is interesting to note how the code inside the package `de.eq3.ble.android.api.command` and the content of the files concerning the management of notifications do not involve parameters related to the mode in which the radiator valve is.

2.2.5 Other Useful Guides

Logging With An Emulator

As already mentioned and depending on your needs, there are various advantages in using an emulator. Among these:

- do **not** need a physical device to do the job
- possibility to easily have a *specific OS* or other features (e.g. root)
- do not put your *privacy* at risk by installing applications from untrusted sources
- *sandboxing* (for example through the use of Firejail)

The only "obstacle" that arises in the use of an emulator is due to the Bluetooth communication. However, it *should* be possible to perform logging and run the application through a **virtual machine** (like Virtualbox) and a **BLE usb dongle**.

This post can be a good point of reference. At the moment we have not been able to test this solution because the dongle we have available is not recognized by the virtual machine.

Similar Projects

Below we present some references concerning the reverse engineering of BLE devices. They are *more focused on the network traffic* than on the Android application, but they can still be useful because they deal with specific devices.

- Syska Smartlight Rainbow LED bulb
- MiPow Playbulb Candle
- Smart Bulb Colorific! light bulb
- NO 1 F4 Smart Band

- Fitbit BLE protocol
- Mikroelektronika Hexiwear

2.3 Protocol Description

2.3.1 Eq3 Eqiva Protocol

This section tries to describe in its entirety the application protocol used by the *Eq3 Eqiva* radiator valves discussed in the *Introduction*. It provides information on the BLE characteristics used, on the composition of commands and notifications, and shows the *Extended BNF*.

The protocol exploits two methods of communication:

1. sending commands to the valve and receiving the respective notification within a short time frame
2. receiving asynchronous notifications

All values exchanged contain the information necessary for their interpretation, therefore it can be considered a **stateless** protocol.

A detailed description will be provided below.

BLE Service And Characteristics

The protocol is based on two characteristics, one for the commands and one for the notifications. Both are part of the same service, identified by the UUID `3e135142-654f-9090-134a-a6ff5bb77046`.

"Send command" characteristic

- **Property:** read/write
- **UUID:** `3fa4585a-ce4a-3bad-db4b-b8df8179ea09`
- **Handle:** `0x0411`

"Notification" characteristic

- **Property:** read/write/notify
- **UUID:** `d0e8434d-cd29-0996-af41-6c90f4e0eb2a`

- **Handle:** `0x0421`

Client Characteristic Configuration Descriptor (CCID)

In general, the CCID, is an optional characteristic descriptor that defines **how** the characteristic may be configured by a specific client (recall: the client is the *central device*). Each client has its own instantiation of the Client Characteristic Configuration . Reads of the Client Characteristic Configuration only shows the configuration for that client and writes only affect the configuration of that client. The characteristic descriptor value is a bit field. When a bit is set, that action shall be enabled, otherwise it will not be used.[1]

In our case, the descriptor exists and has UUID `00002902-0000-1000-8000-00805f9b34fb`. Even if it's not used in this work, its role is fundamental in other contexts, including the development of applications for mobile devices. It allows to activate the *reception of notifications* by setting a bit value to `1` through a write operation.

Note: All the identification codes are made up of *128 bit*. This shows that these are services and characteristics not provided by the Bluetooth LE specification but made by the manufacturer.

Extended Backus-Naur Form

The Backus-Naur Form is a formalism frequently used in the description of the *syntax* and *grammar* of protocols and languages. The "Extended"[2] version will be used below. It is designed for a clearer and more compact representation and is now universally recognized.

Remember that:

- the symbol `|` indicates possible *alternatives*
- the symbol `*` indicates the number of *repetitions*
- `[ ]` identify optional symbols

Protocol Description

```
protocol = command | notification;
```

Command Description

```
command =
    set-date-time | set-temp | set-comfort | set-reduced | modify-comf-
↪reduced
    | boost | auto | manual | holiday | lock | create-profile | read-
↪profile
```

(continues on next page)

[1] Bluetooth Core Specification 5.0, Volume 3, Part G, Section 3.3.3.3
[2] ISO/IEC. Extended Backus–Naur Form, 1996

(continued from previous page)

```
    | window-mode | set-offset;

set-date-time = '03', year, month, day, hour, minutes, seconds;

set-temp = '41', temperature;

set-comfort = '43';

set-reduced = '44';

modify-comf-reduced = '11', temperature, temperature;

boost = '45', (on | off);

auto = '4000';

manual = '4040';

holiday = '40', temperature-128, day, year, hour-and-minutes, month;

lock = '80', (on | off);

create-profile = '10', day-of-week, interval, 6*[interval];

read-profile = '20', day-of-week;

window-mode = '14', temperature, window-minutes;

set-offset = '13', offset-range;
```

Notification Description

```
notification = status-notification | profile-notification;

status-notification = '02', '01', valve-state, [holiday-parameters];

profile-notification = success-modify, profile-read;

valve-state = mode, byte, '04', temperature;

holiday-parameters = day, year, hour-and-minutes, month;

success-modify = '0202', day-of-week;

profile-read = '21', day-of-week, 7*interval;
```

Generic Types

```
on = '01';

off = '00';

year = ('0' | '1' | '2' | '3' | '4' | '5' | '6'), hexdigit;

month =
    '0', ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |␣
↪'A' |
    'B' | 'C');

day = ('0' | '1'), hexdigit;

hour = ('0' | '1'), hexdigit;

minutes = ('0' | '1' | '2' | '3'), hexdigit;

seconds = ('0' | '1' | '2' | '3'), hexdigit;

temperature-128 = ('8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'),␣
↪hexdigit;

temperature = byte;

hour-and-minutes = byte;

day-of-week = '0', ('0' | '1' | '2' | '3' | '4' | '5' | '6');

interval = temperature, byte;

window-minutes =
    '0', ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |␣
↪'A' |
    'B' | 'C');

mode = ('0' | '2'), ('8' | '9' | 'A' | 'C' | 'D' | 'E')

offset-range =
    '0', ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |␣
↪'A' |
    'B' | 'C' | 'D' | 'E')

byte = hexdigit, hexdigit

hexdigit =
```

(continues on next page)

(continued from previous page)

```
    '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' |␣
↪'B' |
    'C' | 'D' | 'E' | 'F'
```

Commands

Commands must be sent to the valve using the hexadecimal system, so the values shown below have already been converted. Alphabetic characters can be indifferently used in uppercase or lowercase.

Note that the valve, and consequently also the CalorBT application, is capable of handling only temperatures whose **decimal part is rounded** to `.0` or `.5`.

Set Current Date And Time

Tells the valve the current date and time. In general it is used to **synchronize with the device**. The notification returned as a result of its execution makes it possible to obtain information about the state.

The command consists of `7 bytes`:

```
byte 0: 03
byte 1: year % 100
byte 2: month
byte 3: day
byte 4: hour
byte 5: minutes
byte 6: seconds
```

Notes

- the `%` symbol represents the modulo operation
- months and days are calculated starting from 1. As a result, both the month of January and the first day of each month will be identified by the value `0x01`

Example

The date and time 25/05/2016 11:27:28 become `03 10 05 19 0B 1B 1C`. At the same way 29/08/2016 09:55:20 becomes `03 10 08 1D 09 37 14`.

Select Temperature (manual)

Activates the selected temperature.

The command consists of `2 bytes`:

```
byte 0: 41
byte 1: temperature * 2
```

Example

Setting the temperature to 18°C is done with command `4124`. This is because 18*2 = 36 = `0x24`. In the same way, to set the valve at 20.5°C it is necessary to send `4129`.

Select Comfort Temperature

Activates the comfort temperature. To modify its default value use *this* command.

The command consists of `1 byte`:

```
byte 0: 43
```

Select Reduced Temperature

Activates the reduced temperature. To modify its default value use *this* command.

The command consists of `1 byte`:

```
byte 0: 44
```

Set Comfort And Reduced Temperature

Changes the default *comfort* and *reduced* temperature values within the valve settings.

The command consists of `3 byte`:

```
byte 0: 11
byte 1: new_comfort_temperature * 2
byte 2: new_reduced_temperature * 2
```

Example

To set the default values for comfort and reduced temperatures to 23°C and 18.5°C respectively, the command `112E25` must be sent to the valve. This is because 23 * 2 = 46 = `0x2E` while 18.5 * 2 = 37 = `0x25`

Start/Stop Boost Mode

Starts or stops the *boost mode* on the valve.

The command consists of `2 byte`:

```
byte 0: 45
byte 1: 01 on // 00 off
```

Select Auto Mode

Activates the automatic mode on the valve. The temperature will reflect the one selected through the weekly schedule.

The command consists of `2 byte`:

```
byte 0: 40
byte 1: 00
```

Select Manual Mode

Activates the manual mode on the valve. The temperature must be selected using the command already shown.

The command consists of `2 byte`:

```
byte 0: 40
byte 1: 40
```

Select Holiday Mode

Activates the holiday mode on the valve. To be activated, it requires: the temperature to be kept and the *end* date and time.

The command consists of `6 byte`:

```
byte 0: 40
byte 1: (temperature * 2) + 128
byte 2: day
byte 3: year % 100
byte 4: (hour*2) + (minutes/30)
byte 5: month
```

Notes

- the `%` symbol represents the modulo operation
- months and days are calculated starting from 1. As a result, both the month of January and the first day of each month will be identified by the value `0x01`
- minutes can only be programmed in half-hour intervals (i.e. XX:00 or XX:30), so the value of `minutes/30` will always be equivalent to 0 or 1.

Example

To maintain the temperature at 17.5°C up to 8.00pm on 10/09/2017 the command is `40 A3 0A 11 28 09`. The byte `0xA3` is derived from the computation of (17.5 * 2) + 128 = 163 = 0xA3, while byte `0x28` was calculated through the selected time as (20 * 2) + (00/30) = 40 + 0 = `0x28`.

Enable/Disable Command Block

It allows to lock the physical buttons on the valve. Note that it allows however to manage the valve through the application.

The command consists of `2 byte`:

```
byte 0: 80
byte 1: 01 on // 00 off
```

Set Temperature Offset

Allows to set a temperature offset in a range between -3.5°C and +3.5°C.

The command consists of `2 byte`:

```
byte 0: 13
byte 1: (temperature * 2) + 7
```

Change Window Mode Settings

Allows to set the duration and the temperature to keep when the window mode takes over. The *window mode* is activated automatically when the valve detects a significant temperature drop.

The command consists of `3 byte`:

```
byte 0: 14
byte 1: (temperature * 2)
byte 2: (minutes / 5)
```

Notes

- minutes can only assume values that are multiples of 5, so the final content of byte 2 will be between `0x00` and `0x0C`

Example

To change the window mode settings to 12°C for the duration of 15 minutes it is necessary to send the command `14 18 03`. Indeed 12*2 = 24 = `0x18` and 15/5 = 3 = `0x03`.

Daily Profile Request

It requires data relating to the schedule of a given day. The information is received as a *notification*.

The command consists of `2 byte`:

```
byte 0: 20
byte 1: day of the week
```

Notes

- the days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*)

Set Daily Profile

Set the schedule for a given day of the week. It is necessary to choose a *base temperature* and it is possible to modify it for **at most three time intervals**. If a profile is already present for the chosen day, it will be replaced.

The command consists of at most `16 byte`:

```
byte 0: 10
byte 1: day of the week
[byte 2-15]: a sequence of at most seven pairs of bytes
```

In each pair `(XX,YY)`:

- `YY` is the **time**, coded as *(minutes/10)*, up to which to maintain the temperature declared in *XX*
- `XX` represents the temperature to be maintained until then, codified as *(temperature*2)*

Notes

- the entire sequence of bytes [2-15] must allow to deduct the temperature to be maintained at any moment of the day
- any unnecessary (*because in excess*) pairs of bytes can be kept at zero or omitted
- the number of minutes in *(minutes/10)* is calculated from the beginning of the day (00:00)
- the days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*)

Example

We want to program the valve so that every Tuesday maintains a base temperature of 17°C and automatically sets itself at:

- 20°C in the range 10:00-12:30
- 19°C in the range 12:30-14:00
- 20°C in the range 15:00-17:00

The command to be sent is `10 03 22 3C 28 4B 26 54 22 5A 28 66 22 90 00 00`, built in the following way:

```
byte 0: 10 (default value)
byte 1: 03 (tuesday = 0x03)
byte (2,3): 22 3C (17°C base temperature up to 10:00)
byte (4,5): 28 4B (20°C up to 12:30)
byte (6,7): 26 54 (19°C up to 14:00)
byte (8,9): 22 5A (17°C up to 15:00)
byte (10,11): 28 66 (20°C up to 17:00)
byte (12,13): 22 90 (17°C base temperature up to 24:00)
byte (14,15): 00 00 (unnecessary, can be omitted)
```

Notifications

As already discussed, notifications are sent from the radiator valve to the central device in order to report to the user the **status** of the device or **the outcome of an operation**.

Also in this case the values are already converted into the hexadecimal numerical system.

Status Notif. (auto/manual mode)

They occur after the execution of any command if the valve is in automatic or manual mode. Note that another type of notification is received after the *read profile* or *write profile* command.

The notification consists of `6 byte`:

```
byte 0: 02
byte 1: 01
byte 2: XY (see below)
byte 3: valve open state in % (from 0x00 to 0x64)
byte 4: undefined (battery level?)
byte 5: (temperature * 2)
```

In the **second byte**:

- `X` indicates if the physical key block is active:

```
X=0 keypad unlocked
X=1 locked due to open window detection
X=2 locked due to manual lock enabled
X=3 locked due to open window detection && manual lock enabled
```

- `Y` indicates the active mode on the valve:

```
Y=8 auto mode
Y=9 manual mode
Y=A holiday mode
Y=C boost mode. at the end it returns to automatic mode
Y=D boost mode. at the end it returns to manual mode
Y=E boost mode. at the end it returns to holiday mode
```

Example

If the valve is in automatic mode, set to 20°C without physical buttons locked, by executing the "Activate boost mode" command, the notification `02 01 0C XX XX 28` is received. The byte `0x0C` supplies the information related to the unlock status and the mode in use, while the last byte `0x28` corresponds to twice the set temperature.

According to the same logic, by setting the valve in manual mode, with the physical buttons locked and setting the temperature to 21.5°C, the notification takes the value `02 01 29 XX XX 2B`.

Status Notif. (holiday mode)

They occur after the execution of any command if:

- the valve is in **holiday mode**
- the valve is in **boost mode** and at its end it will return to holiday mode

Note that another type of notification is received after the *read profile* or *write profile* command.

The notification consists of `10 byte`:

```
byte (0-5): same as in previous section (3.4.1)
byte 6: end_holiday_day
byte 7: end_holiday_year%100
byte 8: (end_hour*2) + (end_minutes/30)
byte 9: end_holiday_month
```

Notes

- minutes can only be programmed in half-hour intervals (i.e. XX:00 or XX:30), so the value of `end_minutes/30` will always be equivalent to 0 or 1.

Example

By activating the holiday mode with the physical buttons locked, temperature at 20°C and the end date set to 18:30 on 14/12/2016, the valve provides the following notification: `02 01 2A XX XX 24 0E 10 25 0C`.

The first six bytes are consistent with what was stated in the previous section: `0x2A` declares that the holiday mode is active. Of the remaining four bytes, `0x0E`, `0x10` and `0x0C` indicate the *day*, *year*, and *month* respectively. Finally, the `0x25` encodes the time "18:30" according to the method described: (18*2) + (30/30) = 36 + 1 = 37 = `0x25`.

Profile Notif. (modify)

They appear after the *Set Daily Profile* command has been sent. They confirm the execution.

The notification consists of `3 byte`:

```
byte 0: 02
byte 1: 02
byte 2: modified_day
```

Notes

- the days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*)

Profile Notif. (request)

They appear after the *Daily Profile Request* command has been sent. They provide all the information necessary to identify temperatures and time ranges for the selected day.

The notification consists of `16 byte`:

```
byte 0: 21
byte 1: day_of_the_week
(byte 2-15): a sequence of seven pairs of bytes (see below)
```

In each pair `(XX,YY)`:

- `YY` is the **time**, coded as *(minutes/10)*, up to which to maintain the temperature declared in *XX*
- `XX` represents the temperature to be maintained until then, codified as *(temperature*2)*

Notes

- the entire sequence of bytes (2-15) allows to deduct the temperature to be maintained at any moment of the day
- any unnecessary (*because in excess*) pairs of bytes will have the value `XX=base_temperature` and `YY=0x90`

- the days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*)

Example

We program the valve so that every Monday maintains a base temperature of 17°C and automatically sets itself at:

- 21°C in the range 06:00-06:00
- 21°C in the range 17:00-23:00

We request the profile and we get the notification: `21 02 22 24 2A 36 22 66 2A 8A 22 90 22 90 22 90`, built as follows:

```
byte 0: 21 (default value)
byte 1: 02 (Monday = 0x02)
byte (2,3): 22 24 (17°C up to 06:00)
byte (4,5): 2A 36 (21°C up to 09:00)
byte (6,7): 22 66 (17°C up to 17:00)
byte (8,9): 2A 8A (21°C up to 23:00)
byte (10,11): 22 90 (17°C up to 24:00)
byte (12,13): 22 90 (unused)
byte (14,15): 22 90 (unused)
```

2.3.2 Laica PS7200L Protocol

The PS7200L is a BLE scale produced by the italian company Laica. In addition to measuring weight, it allows the calculation of:

- fat percentage
- water percentage
- skeletal muscle mass percentage
- skeletal system weight
- base metabolism
- body mass index

To use all the features of the scale, the company provides an application, called *laicabodytouch*, for Android and iOS devices. The application receives the data from the scale, keeps track of it and shows it to the user through simple graphs.

Fig. 26: Laica PS7200L BLE Scale

It is interesting to observe the **permissions** required by the application to perform such a “simple” task. Some of them are quite *ambiguous* and are not essential to use Bluetooth[4]. For example, some required permissions are:

- retrieve running apps
- find/add/remove accounts on the device
- read phone status and identity
- view Wi-Fi connections

Note: At present, the PS7200L protocol has been reverse-engineered only in a small part. However, it has been included in this guide because it shows aspects that have not been dealt within other sections.

BLE Communication

After analyzing the log files using the methods described in the *Logging Via Android* section, it becomes clear that the application **does not send commands** to the scale.

The operation principle is based on the fact that when a person is on the scale, this starts broadcasting *advertising packages*. These packages contain all the information about the device and the person using it. For this reason, there is not even a pairing procedure between the central device and the BLE device.

Through the advertising packages, the scale provides the application **two essential information**:

1. *the weight* in Kg
2. a value used to derive *all the other parameters* (such as *fat%*, *water%*, etc..)

Note: To achieve the goal of point 2 (derive all the parameters), the application also uses three *manually* entered data: gender, age and height.

[4] Android Bluetooth Permissions

Advertising Packets Content

The structure of an advertising package is clearly described in the Bluetooth specifications[3]. In addition to a field that indicates the MAC address of the device who sent the package, there is an **Advertising Data** field that basically represents the payload.

Referring to the way *Wireshark* presents the packages (see *Wireshark log of Advertisement packets*), the data that compose the communication protocol between the two devices are the *12 Byte* under `Advertising Data > Manufacturer Specific > Data`. Their meaning is as follows:

```
..
byte (2,3): weight*10 (in Kg)
byte (4,5): used to derive other parameters (fat%, water%, etc..)
..
```

At the moment it is not clear how the value of the *bytes (4,5)* can lead to the calculation of all the other data. Even decompiling the Android application the analysis is blocked by the fact that these two bytes are used as input to a function called `getHealth()` in the **proprietary library** `libyohealth.so`.

We decompiled the library using `objdump` (with an ARM toolchain) and the RetDec decompiler, which provides output in C language. However, for obvious reasons, the results are not easily readable. The outputs obtained from the decompilation process are available here.

2.4 Script Creation

This section wants to introduce some useful tools to search for Bluetooth devices and to communicate with them. In particular, we consider the tools provided by the *BlueZ* stack. Then we explain, as an example, some details on the scripts created for the management of the *Eq3 Eqiva* radiator valves discussed in the *Introduction*.

2.4.1 Bluez Stack

BlueZ is the offical Linux Bluetooth protocol stack and is part of the Linux kernel since version 2.4.6 was released[4]. It provides the necessary modules to manage *both classic and low energy* Bluetooth devices[5]. Along with these modules, there is a series of command-line tools (*bluez-tools*) that interact with the main core.

[3] Bluetooth Core Specification 5.0, Volume 2, Part E, Page 1193
[4] BlueZ - Common Questions
[5] Ubuntu Bluez Documentation

Debian provides all the necessary packages in its repositories. The installation can be performed through the `apt` package manager:

```
$ sudo apt install bluez bluez-tools
```

Many tools are available. Below we describe the use of those needed to send a command to a *BLE* device and to search for its MAC address.

Hcitool

As specified in its manual[6], `hcitool` is used to configure Bluetooth connections and send some special command to Bluetooth devices. It allows, among other things, to send HCI commands, make connections and manage the authentication phases.

Furthermore, it is able to scan for both BLE and non-BLE devices. This allows us to identify the **MAC address** of the device we want to work on. However, it's not possible to search both types of devices simultaneously:

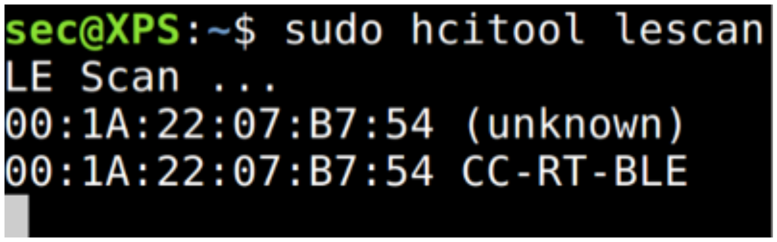
- to start a classic scan, use the command:

  ```
  $ sudo hcitool scan
  ```

- to start a BLE scan, use the command:

  ```
  $ sudo hcitool lescan
  ```

The following image shows the result of a BLE scan. Each line contains the MAC address of the identified device followed by its name (if readable).



```
sec@XPS:~$ sudo hcitool lescan
LE Scan ...
00:1A:22:07:B7:54 (unknown)
00:1A:22:07:B7:54 CC-RT-BLE
```

Fig. 27: Scan for BLE devices

Note: The `scan` and `lescan` commands ignore advertising packages received from devices they already know about. To prevent those packages from being ignored (in some cases it may be necessary) use the `--duplicate` parameter.

[6] hcitool manpage

Hcidump

Hcidump is a tool that can read the contents of the HCI packets sent and received by any Bluetooth device.

Despite the BlueZ stack integrates this tool (*since version 5*), it requires to be installed separately. As always, in Debian this can be done using the `apt` package manager:

```
$ sudo apt install bluez-hcidump
```

Among other things, we will use this software to **read the contents of the advertising packages** during the scan phase. To do this:

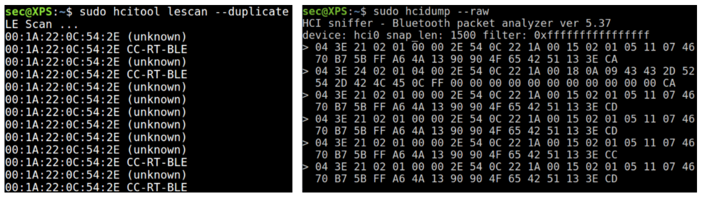
1. start a *scan* through *hcitool*:

   ```
   $ sudo hcitool lescan --duplicate
   ```

2. while *hcitool* is running, start `hcidump`:

   ```
   $ sudo hcidump --raw
   ```

The `--raw` parameter allows to obtain data in the "original" format. Other parameters (`--hex`, `--ascii`) can be used to get them in other formats. Refer to the manual[8] for these details.



```
sec@XPS:~$ sudo hcitool lescan --duplicate
LE Scan ...
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E CC-RT-BLE
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E CC-RT-BLE
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E CC-RT-BLE
00:1A:22:0C:54:2E (unknown)
00:1A:22:0C:54:2E CC-RT-BLE
```

```
sec@XPS:~$ sudo hcidump --raw
HCI sniffer - Bluetooth packet analyzer ver 5.37
device: hci0 snap_len: 1500 filter: 0xffffffffffffffff
> 04 3E 21 02 01 00 00 2E 54 0C 22 1A 00 15 02 01 05 11 07 46
  70 B7 5B FF A6 4A 13 90 90 4F 65 42 51 13 3E CA
> 04 3E 24 02 01 04 00 2E 54 0C 22 1A 00 18 0A 09 43 43 2D 52
  54 2D 42 4C 45 0C FF 00 00 00 00 00 00 00 00 00 00 00 CA
> 04 3E 21 02 01 00 00 2E 54 0C 22 1A 00 15 02 01 05 11 07 46
  70 B7 5B FF A6 4A 13 90 90 4F 65 42 51 13 3E CD
> 04 3E 21 02 01 00 00 2E 54 0C 22 1A 00 15 02 01 05 11 07 46
  70 B7 5B FF A6 4A 13 90 90 4F 65 42 51 13 3E CD
> 04 3E 21 02 01 00 00 2E 54 0C 22 1A 00 15 02 01 05 11 07 46
  70 B7 5B FF A6 4A 13 90 90 4F 65 42 51 13 3E CC
> 04 3E 21 02 01 00 00 2E 54 0C 22 1A 00 15 02 01 05 11 07 46
  70 B7 5B FF A6 4A 13 90 90 4F 65 42 51 13 3E CD
```

Fig. 28: Hcidump (*right*) running while scanning (*left*)

The *image above* shows the execution of the previous commands. The terminal on the right lists the intercepted HCI packets. They are all advertising packages, because the first bytes are `04 3E`. As indicated by the Bluetooth specifications, `04` stands for *HCI Event* while `3E` means *LE Advertising Report*.

An example of the use of `hcitool` and `hcidump` is given in the *Laica PS7200L Scripts* section.

Gatttool

Gatttool is designed for Bluetooth Low Energy. It therfore works exploiting the concept of *GATT* and its ATT protocol, which is adopted only by BLE devices. Basically, it allows you to connect to a device, discover its characteristics, write/read attributes and receive notifications. Optionally, it can also be used in *interactive mode* through a CLI.

[8] hcidump manpage

Possible uses are described in the manual[7] and through the command `gatttool --help`. Below we show how to write a value on a characteristic, which is what we want to do in order to send commands to BLE devices.

To this end, you need:

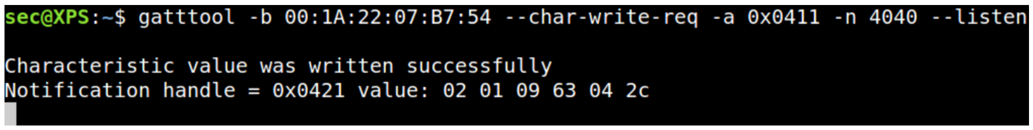
- the **MAC address** of the target device
- the *Handle* that identifies the characteristic.

Given this data, the following command sends the value represented by `<value>` to the device:

```
$ gatttool -b <mac_address> --char-write-req -a <handle> -n <value> --
↪listen
```

Note: The `--listen` parameter requires `gatttool` to wait for a notification. However, this puts the process in a waiting state until it is *manually* terminated.

The *following image* shows the output obtained from the command execution. As you can see, the characteristic has been written correctly and information about the received notification is shown.



```
sec@XPS:~$ gatttool -b 00:1A:22:07:B7:54 --char-write-req -a 0x0411 -n 4040 --listen

Characteristic value was written successfully
Notification handle = 0x0421 value: 02 01 09 63 04 2c
```

Fig. 29: Gatttool Characteristic Write

2.4.2 Eq3 Eqiva Script

An english translation of the software is available here (while here is the original) and consists of:

- a series of functions useful to interact with a single valve
- some scripts for the simultaneous management of multiple valves

Everything has been developed and tested on a *bash shell* and this should ensure portability on most *Unix* systems without having to make many changes. However, the *BlueZ stack* dependency limits its use to GNU/Linux operating systems.

Updated:

We have created an updated version of the scripts. It corresponds to the one implemented in the eq3eqiva .deb package. The next sections will refer to the “original” (old) version. Changes and additions of the new implementation will be reported with the **Updated** tag.

[7] gatttool manpage

GPLv3 License

As mentioned in the *Introduction*, the project is (among other things) aimed at providing the possibility to integrate these valves into free home automation systems.

The code is then released under the **GPLv3** license (GNU General Public License), the conditions of which have to guarantee the *four fundamental freedoms*[1] defined by the Free Software Foundation:

1. The freedom to run the program as you wish, for any purpose (freedom 0).
2. The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
3. The freedom to redistribute copies so you can help others (freedom 2).
4. The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

Basically, it is a noticeably **copyleft** license: anyone who wants to distribute copies of a software bound to these conditions, whether free or behind the payment of a price, is obliged to recognize the same rights that it received to the recipient. He must also guarantee access to the source code.

Single Valve Management

Everything necessary to manage a single valve is contained in two files:

- `basic_functions.sh` deals with *sending* and *receiving* data, translating notifications and also contains some frequently used functions.
- `valve_commands.sh` deals with the syntactic/semantic *composition* of each command

Both files are commented and describe each function, therefore only some relevant aspects will be discussed below.

It is interesting to note that, although the *CalorBT* application requires a **pairing procedure** in order to be able to communicate with the valve, this procedure is not necessary outside the Android/iOS context. This is clearly a **security flaw** because it allows communication with the device while knowing only the MAC address, which can easily be obtained through external tools such as the already described *hcitool*

basic_functions.sh File

send_command()

[1] The Free Software Foundation. What is free software?

```
input: <device_address> <command>
output: value of the notification received after execution
```

It writes the value of the `command` argument on the *"send command" characteristic*, thus causing the execution of the corresponding command.

The transmission is based on the use of the *Gatttool* tool, as discussed in the *previous section*, through the following line of code:

```
output=$(timeout $TIMEOUT_SEC gatttool -b $1 --char-write-req -a␣
↪0x0411 -n $2 --listen)
```

The parameters `-b` and `-n` allow to specify respectively the address of the device and the value to send; at the time of execution they will be replaced by `<device_address>` and `<command>`. The use of `-listen` puts the tool in a locked state, waiting for one or more notifications from the counterpart.

However, there is no way to indicate a temporal term to this condition, and it is necessary to use `timeout $ TIMEOUT SEC` to store what is received in the output variable after a certain period of time and move on to the next instruction. The `TIMEOUT SEC` variable is defined in the `config.sh` file and will be detailed in the *Field Test section*.

---

parse_return_value()

```
input: <notification_value>
output: notification translated into readable content
```

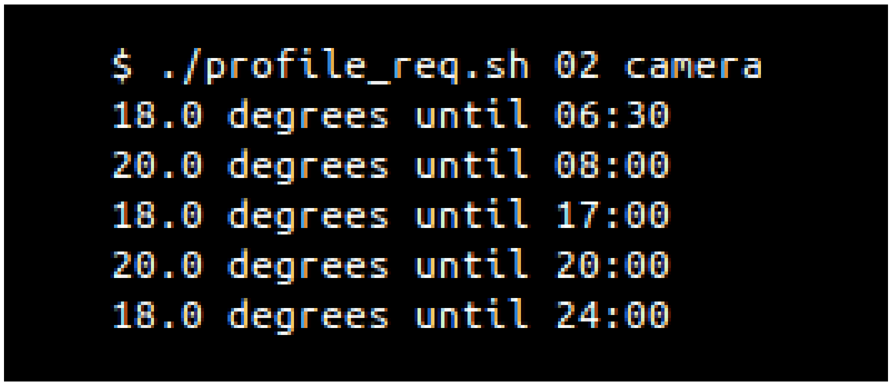
It translates the notification content and brings it to the standard output through a series of `echo` commands. The parameter `<notification_value>` must be composed of the received bytes, **each separated by a space**: the same format used by *Gatttool*.

The translation is performed by interpreting the value of each byte consistently with the information in section *Notifications*. If it is a *Holiday Mode notification*, the parsing is done with the help of the `parse_holiday_params()` function, which is also contained in `basic_functions.sh`, using bytes 7, 8, 9 and 10.

As an example, the `profile_req.sh` script allows to request information on a day schedule using the appropriate command. The output produced is an example of using the `parse_return_value()` function. The *above figure* shows the translation of the notification received following the request relating to the day of Tuesday (indicated by the parameter `02`) to the valve called "camera".

---

calculate_temp()



```
$ ./profile_req.sh 02 camera
18.0 degrees until 06:30
20.0 degrees until 08:00
18.0 degrees until 17:00
20.0 degrees until 20:00
18.0 degrees until 24:00
```

Fig. 30: profile_req.sh example

```
input: temperature in decimal base
output: (temperature*2) in hexadecimal base and rounded
```

It allows to encode the temperature according to the format used by the valve. The result is obtained by multiplying by 2 and then rounding to a value equal to `XX.0` or `XX.5`.

Notes

This function must not be used for the *Select Holiday Mode* command. It uses the rule `(temperature*2)+128`. For this purpose there is the function `calculate_temp_128()`, which is also contained in `basic_functions.sh`.

---

search_by_name()

```
input: <valve_name> <file_name>
output: valve MAC address (-1 if it does not exists)
```

It allows to find the **MAC address** of a single device.

For this purpose, each valve is identified by a user-assigned name within the file `file_name`. Within this file, each valve must be represented on a new line according to the format: `NAME/ADDRESS`. The function does not distinguish between uppercase and lowercase letters; therefore, the line `valve1/00:11:22:33:44:55` is completely equivalent to `VALVE1/00:11:22:33:44:55`.

The file is scanned line by line: if there are duplicates, the *first occurrence* will be selected.

Updated:

The following functions have been included in the updated version.

check_bt_status()

```
input: -
output: an error message if Bluetooth isn't active
```

Refer to *this section* for more details.

validate_mac()

```
input: <MAC_address>
output: 0 if valid, -1 otherwise
```

Check if a MAC address is syntactically valid. To be *valid*, an address must be composed of a succession of six pairs of hexadecimal values, separated by "`:`". The check is done through a regular expression, using the following code.

```
#check if $1 is a valid mac address
if [[ "$1" =~ ^([a-fA-F0-9]{2}:){5}[a-fA-F0-9]{2}$ ]]; then
    echo "0"
else
    echo "-1"
fi
```

In particular, it requires that a pair `{2}` of hexadecimal values `[a-fA-F0-9]` be repeated five times `{5}`, each time followed by "`:`". There must then be a sixth pair, again identified by `[a-fA-F0-9]{2}`, which must not be followed by the separator "`:`" (because it is the last pair).

valve_commands.sh File

As already disclosed, `valve_commands.sh` manages the *syntactic composition* of every possible command and requires its execution through the following functions. Note that all functions **automatically round the entered temperature** to values of the type `XX.0` or `XX.5`.

- **send_init() `<device_address>`** Send the current date and time, automatically calculated through the command `date`. It is not essential, but it is useful to start the communication in order to guarantee the synchronization between the central device and the valve and to receive a notification that reports the status.
- **boost_mode() `<device_address>`** Causes *boost mode* activation.
- **stop_boost_mode() `<device_address>`** Causes *boost mode* deactivation.
- **auto_mode() `<device_address>`** Activate the automatic mode and adjust the temperature accordingly (as selected in the weekly schedule).
- **manual_mode() `<device_address>`** Activate the manual mode.
- **set_temperature() `<device_address>` `<temperature>`** Set the temperature to the value indicated by the second parameter.

- **set_comfort_reduction_temp() `<device_addr> <comf_temp> <red_temp>`** Changes the *"comfort temperature"* and *"reduced temperature"* values within the valve settings.
- **holiday_mode() `<device_address> <DD/MM/YYYY> <hh:mm> <temperature>`** Activate the holiday mode; therefore maintains the same temperature until the end indicated by the parameters.
- **read_profile() `<device_address> <day>`** Require the daily schedule. Days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*)
- **set_profile()** `<device_address> <day> <int1> [int2] [int3] [int4] [int5] [int6] [int7]`

  Set the daily schedule. Days of the week are counted starting from Saturday (`00` is *Saturday*, .., `06` is *Friday*). Each interval `intX` must be in the form `TEMPERATURE/hh:mm` and together they must guarantee coverage for the whole day following the guidelines in the section *Set Daily Profile*. As a result, the intervals 2-7 may turn out to be unnecessary and certainly the last one specified must have the time `24:00` or `00:00`.
- **lock() `<device_address>`** Locks the physical keys on the valve.

  NOTE: This does not prevent interaction through Bluetooth.
- **unlock() `<device_address>`** Unlocks the physical keys on the valve.
- **set_window() `<device_address> <temperature> <duration>`** Set *window mode* temperature and duration.
- **set_offset() `<device_address> <temperature>`** Set the offset temperature. It must be between -3.5 and +3.5.

The following is a general example of how these functions work, since they are all quite similar.

```
1  read_profile() {
2      if [ $2 -lt 0 -o $2 -gt 6 ]; then
3          echo "Week goes from 00 (saturday) to 06 (friday)."
4          return
5      fi
6  
7      day=$(printf "%02x" $2) # $2 = day
8      echo $( send_command $1 20$day ) # $1 = device_address
9  }
```

The first part, represented here by lines 2-7 but not always necessary, checks the correctness of the inputs and calculates the coding of the parameters according to the format required by the valve. The second part (*line 8*) sends the command using the *send_command* function and prints the notification received.

Parsing/Translation

As shown in the example, the default behavior does not provide *parsing* and *translation* of what has been received. For this to happen you need to use the *parse_return_value* method, in two possible alternative ways:

1. replacing it with the `echo` command in the last line of each function (*line 8 in the example*):

   ```
   parse_return_value $( send_command $1 20$day )
   ```

2. moving the `parse_return_value()` call outside of the requested function (*read_profile()* in the example), thus obtaining the following code:

   ```
   parse_return_value $( read_profile ....... )
   ```

Multiple Valve Management

The functions described in the previous section are useful to create larger scripts that automate the management of multiple valves. Below are listed the **scripts** made, which want to form a simple guideline in the development of larger projects.

Configuration

For simplicity, using the scripts requires assigning a name chosen by the user to the MAC address of each valve. It is therefore necessary to indicate which will be the file containing the *name-address associations* through the `VALVE_FILE` variable inside the `config.sh` file.

The file referred to by `VALVE_FILE` must be compiled according to the syntax required by the documentation of *search_by_name*, thus using the format `NAME/MAC_ADDRESS`.

MAC addresses can be found as described in the *Bluez Stack* section through the `hcitool lescan` command.

In order to run the scripts you need execution permissions:

```
$ cd /path/to/scripts/directory
$ chmod u+x *.sh
```

Then, to run a script:

```
$ ./script_name.sh parameter1 parameter2
```

Updated:

For each script, except for *profile_req*, you can use the `-p` or `--parse` option to activate the parsing of the received notifications. Otherwise, the value of notifications is shown in hexadecimal base.

Example: `./auto_mode.sh -p bathroom kitchen`

- **auto_mode `<valve_name> [valve_name ...]`** Set the "auto mode" on all the valves identified by the names provided by command line. At least one parameter is required, while the subsequent ones are optional.

  NOTE: the temperatures set on each valve after the execution of the script *are dependent* on how they have been programmed individually.

- **manual_mode `<temperature> <valve_name> [valve_name ...]`** Set the "manual mode" on all the valves identified by the names provided by command line. At least one parameter is required, while the subsequent ones are optional.

- **set_temperature `<temperature> <valve_name> [valve_name ...]`** Rounds the `<temperature>` value and sends it to all the valves identified by the names provided by command line. Requires the first and second parameters, while the subsequent ones are optional.

- **set_all_temp `<temperature>`** Rounds the `<temperature>` value and sends it to all the valves. Names and addresses of the valves to which the data are sent are taken from the file referenced by `VALVE_FILE`.

  NOTE: The file is scanned line by line. The presence of duplicates implies a double execution of the command on the same valve

- **set_profile `<profile_file> <valve_name> [valve_name ...]`** Set profiles for one or more days on all the valves identified by the names provided by command line. The values to be set are supplied via a `profile_file`. It must contain the schedule for one or more days in the following format:

  ```
  day (1=monday, ..., 7=sunday)
  base_temperature
  HH:MM-HH:MM-TEMP (first interval)
  HH:MM-HH:MM-TEMP (secondo optional interval)
  HH:MM-HH:MM-TEMP (third optional interval)
  end
  ```

  NOTE: within the same file, you can specify the schedule for *several days* separating each block by an empty line.

  As an example, the Monday schedule with a base temperature of 18°C and 20°C in the two ranges 06:30-08:00 and 17:00-20:00 is carried out in this way:

  ```
  01
  18
  06:30-08:00-20
  17:00-20:00-20
  end
  ```

  Updated:

  In the new implementation **days** are indicated by their names and not numerically.

They must be written in English in complete or abbreviated form (e.g. “Saturday” is equivalent to “sat”). They are *not case sensitive*.

The previous example becomes:

```
Monday
18
06:30-08:00-20
17:00-20:00-20
end
```

- **profile_req `<day> <valve_name> [valve_name ...]`** It requires and prints the daily schedule for all the valves identified by the names provided by command line. The `<day>` parameter is counted starting from `01` (*Monday*) up to `07` (*Sunday*).

  Updated:

  In the new implementation **days** are indicated by their names and not numerically. They must be written in English in complete or abbreviated form (e.g. “Saturday” is equivalent to “sat”). They are *not case sensitive*.

  Example: `./profile_req Sunday bathroom kitchen`

The operating principle of each script is almost identical. First of all, the presence of the parameters required to run it is checked. Then each parameter representing a device activates the search function of the MAC address. Once this is done, the commands are sent to the valves. The following lines of code, extracted from the `auto_mode.sh` script, clarify the functions used.

```
1  #!/bin/bash
2  . ./valve_commands.sh
3
4  if [ -z $1 ]; then
5      printf "Usage: ./auto_mode.sh <valve_name> [valve_name ...]\n"
6      exit
7  fi
8
9  for name in "$@"; do
10     address=$( search_by_name $name $VALVE_FILE )
11
12     if [ "$address" != "-1" ]; then
13         auto_mode $address
14     else
15         printf "%s valve not found\n" "$name"
16     fi
17 done
```

The inclusion of *valve_commands.sh‘* (*line 2*) makes available all the primitives present in the section *Single Valve Management*. This causes the implicit inclusion of `basic_functions.sh` and `config.sh` and makes usable the *search_by_name* function and the `VALVE_FILE` variable.

Lines 4-7 check for the required parameters and cause the script to exit if the check is not passed. The *for* loop (*lines 9-17*) allows to go by all the names of the valves supplied as an argument to the script. From each of these the MAC addresses are obtained. Now sending the request to the valve is simple and is based on the call to an already known function (*line 13*): `valve_commands.sh` provides the necessary to carry out all the operations made available by the *CalorBT* application.

The part of the code that deals with the transmission of data to the valve is located within the `for` cycle. For this reason the requested command is sent to **one device at a time**, in the order in which the names are supplied to the script at invocation time (see *Sequence Diagram of auto_mode.sh*). The output produced corresponds to the notifications received from time to time, after the execution of each command.

If the address of a valve is not found, a control (*line 12*) causes an error message to be printed (*line 15*). Then starts the search for the next address (if required by the entered parameters).

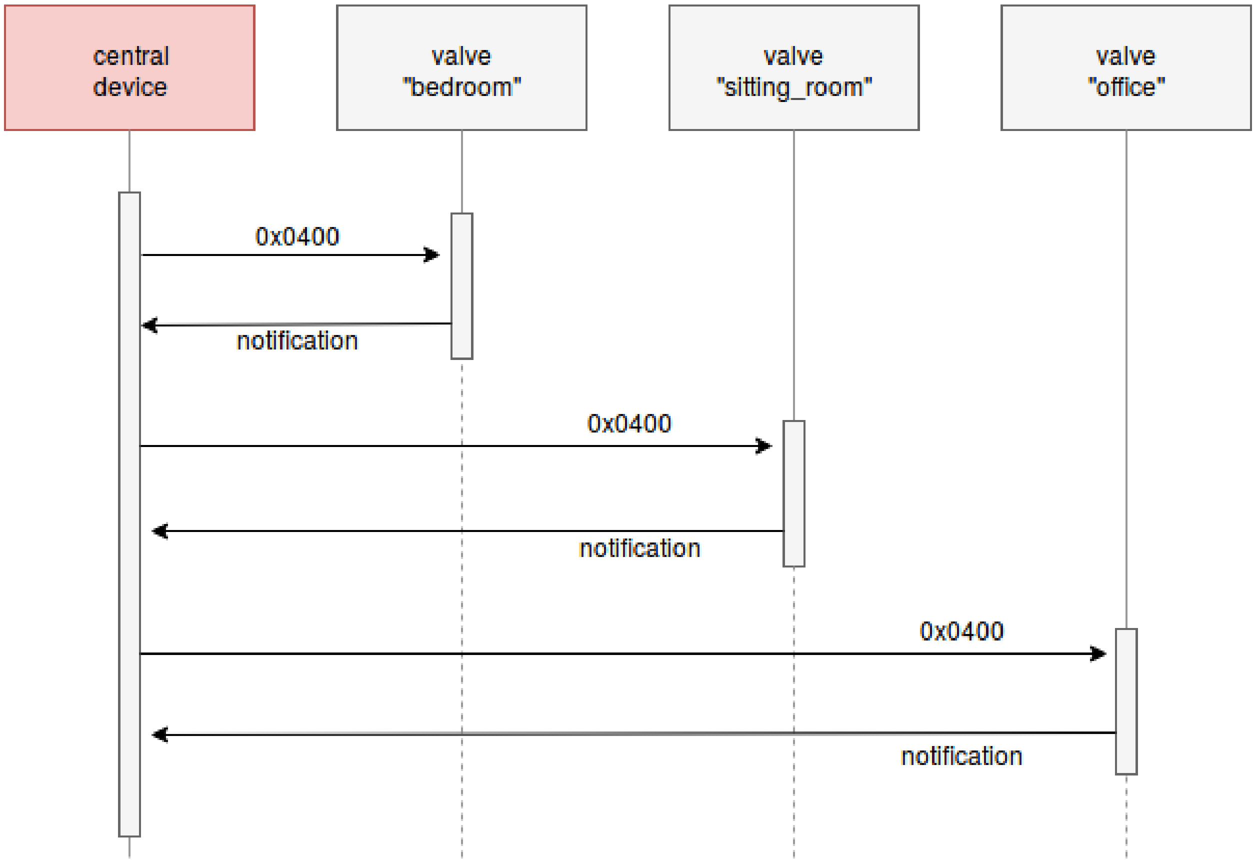


Fig. 31: Sequence Diagram of *auto_mode.sh*

The above *Sequence Diagram*[2] shows the exchange of messages between three valves and the central device during the execution of the `auto_mode.sh` script, activated as follows:

[2] Sequence Diagram - Wikipedia

```
./auto_mode.sh bedroom sitting_room office
```

Correctly, as we have just seen, the value `0x0400` (*Select Auto Mode*) is sent in succession for each valve. The next command is sent only after receiving the notification of the previous one.

The `set_profile.sh` script is the only exception to this behavior, because it has to read the supplied file. This can contain instructions related to the schedule of several days, making it necessary to send **several commands to the same valve**. For this purpose it was decided to minimize the number of calls to the `search_by_name()` function by immediately searching for the requested MAC addresses and storing the result in an array (the `VALVES_ADDR` variable, inside the script).

Updated:

In the new implementation, in each script was added a control on Bluetooth activation, syntax validation of MAC addresses and the possibility to use the `-p` or `--parse` option (see the *previous note*).

The first two points were obtained through the functions:

- *check_bt_status*
- *validate_mac*

The third point was obtained through the following code

```
#check if the -p or --parse option is present
case $1 in
        -p|--parse)
        parsing=1
        shift #discard the argument

        ..  #do things
        ;;

        *)
        parsing=0
        ;;
esac
```

If the first parameter (*represented by $1*) corresponds to `-p` or `--parse`, the variable `parsing` is set to 1 and the parameter already used is discarded through the **shift** command. Now $1 contains the next parameter supplied by the user. In all other cases (*identified by* "`*)`"), `parsing` is set to 0 and no parameters need to be discarded.

As a result, the overall structure of each script is a little changed. The new structure is:

```
#check if the parameter -p is present
.. #code above
```

(continues on next page)

(continued from previous page)

```
3
4  check_bt_status  #check if bluetooth is active
5
6  for name in "$@"; do
7    address=$( search_by_name $name $VALVE_FILE )
8
9    if [[ "$address" != "-1" && $(validate_mac $address) != "-1" ]]; then
10
11     notification=$(auto_mode $address)
12
13     if [[ ! -z $notification && $parsing = 0 ]]; then
14       printf "\n%s: %s\n" "$name" "$notification"
15     elif [[ ! -z $notification && $parsing = 1 ]]; then
16       printf "\n%s:\n" "$name"
17       parse_return_value $notification
18     else
19       #received empty notif
20       printf "\n%s: error. try to increase the timeout or move close to the valve\n" "$name"
21   fi
22   else
23     #mac address error or valve not found
24     printf "\n%s: not found in '%s' or invalid MAC address\n" "$name" "$VALVE_FILE"
25   fi
26 done
```

First of all we check the presence of the `-p|--parse` option and the activation of the Bluetooth (*line 4*). Then there is the usual `for` cycle, in which there are some differences with respect to the *previous implementation*.

The *line 9* not only verifies that the MAC address has been found, but also that it is **correct at syntax level** (through the *validate_mac* function).

Furthermore, in this version the notification received from the valve is saved in the `notification` variable (*line 11*). This is because depending on whether the `parsing` variable is set to `0` or `1` (respectively, *line 13* and *line 15*), the notification must be shown in hexadecimal values (*line 14*) or translated (*line 17*).

The **errors** are substantially divided into two cases:

- something went wrong in the communication and the notification received is empty (*line 20*)
- no valve was found with that name or the MAC address entered by the user is not *syntactically* correct (*line 24*)

In both cases error messages are printed.

Field Test

The information in the previous sections was tested using:

- a Bluetooth 4.0 *Class 2 adapter* on the central device
- *Ubuntu 14.04 LTS* GNU/Linux
- the *4.101 version of the BlueZ stack* (included by default in the chosen OS)

However, the scripts have also been tested on the latest BlueZ versions.

Range And Related Problems

The EQ3 company, which produces the valves, declares a maximum range of 10 meters outdoors (and therefore in the absence of obstacles)[3]. Despite this, the use via smartphone (through the appropriate application) allowed us to reach the distance of 12.5 meters without drops in reception.

The signal's **range and strength** directly influence the content of the `TIMEOUT_SEC` variable, defined in `config.sh`. This variable is used in the *send_command* function. Its value represents the **maximum time (in seconds)** within which it is certain to be able to carry out the following operations:

1. connect to the valve
2. send the command
3. receive the subsequent notification

DISTANCE	AVERAGE TIME	STD. DEV.
2 meters	02.99 sec.	0.44 sec.
4 meters	03.78 sec.	0.59 sec.
6 meters	13.73 sec.	1.80 sec.

The value (time) of `TIMEOUT_SEC` is set by default to *5 seconds*, given the results obtained with the Class 2 adapter used in the tests. As can be seen from the table, this adapter has signal losses even at a distance of 6 meters, making it impossible to deepen the analysis. As a consequence, the most appropriate value was selected with respect to the results obtained under good reception

[3] Bluetooth Smart Radiator Thermostat - EQ3

conditions (*2 and 4 meters*). In all likelihood, these values will not suffer large increases over medium to long distances using a more powerful Bluetooth connector (i.e. *Class 1*).

Note: `TIMEOUT_SEC` is in any case a fundamental parameter that needs to be *manually adapted* depending on the application context. The goal is to find a good compromise between waiting times and errors (due to not receiving the notification within the short time available).

The calculation of the elapsed time between connection, sending and receiving was done using the `time` command in the following way:

```
time gatttool -b $ADDRESS --char-write-req -a 0x0411 -n 4000
```

This sends, by way of example, the *Select Auto Mode* instruction to the valve using *gatttool*. The average times and standard deviations shown in the table result from the execution of this instruction on three different valves. In total, *20 trials* were performed for each distance.

Each test was carried out by the same **starting state**: valves disconnected from the central device. The disconnection automatically occurs 45 seconds after execution of a command.

Parallel Connection

The *gatttool* tool used for the connection does not allow parallel sending of commands. The central device is able to communicate with only *one device at a time*.

Indeed, after a write operation of a characteristic, `gatttool` does not allow the execution of operations on other devices until the actual confirmation of success (in the form of a *notification*) is received. This makes every attempt useless.

Similar Projects

Recently a Python library that allows the use of these valves has been integrated into the home-assistant platform. The library can be used “stand alone” (i.e. without *home-assistant*) through a CLI interface. Here are some differences with respect to our implementation.

Pros:

- Better Bluetooth management, probably thanks to bluepy. It seems **more reliable** when something goes wrong and on average requires less waiting time.
- Provides information on the status of the valve battery.

Cons:

- Requires *python* and *bluepy* (not present in Debian repositories).
- With the CLI interface it is possible to **memorize only one valve** (by exporting the MAC address to an environment variable or by specifying it manually at each command). Through

the `home-assistant` platform apparently you can pair "name-address" (more than one), but it is a feature implemented in `home-assistant` itself.

- There is no way to set up *daily schedules*. It also returns strange values (mixed with correct information) when they are read.

Note: The repository does not provide information about the Eq3 Eqiva protocol: *it is not documented*. Although this is not a real "downside" in use, we think it is important to spread the result of a reverse engineering activity.

2.4.3 Laica PS7200L Script

Scripts related to the *Laica PS7200L* BLE scale are available here. They have been tested on a *bash shell* and the code is released under the *GPLv3 License*.

In addition to the *BlueZ stack*, they require the `hcidump` tool. Refer to the *previous section* for clarification on use and installation.

Basically, the scripts are composed of two files:

- `basic_functions.sh` contains a series of functions to *receive information* from the BLE scale and convert them to a readable format
- `get_weight.sh` automates calls to basic functions in order to get the weight in kg

basic_functions.sh File

check_bt_status()

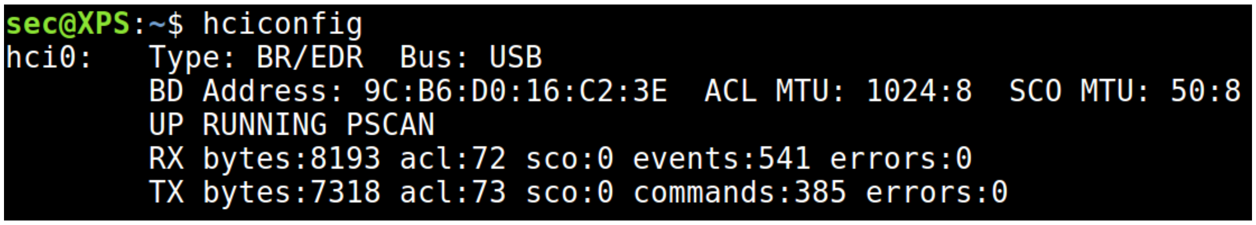
```
input: -
output: an error message if Bluetooth isn't active
```

Check if the Bluetooth is active on the central device. Otherwise, prints an error message and the list of each identified Bluetooth adapter (with its *status*).

For this purpose, the function uses the output generated by the `hciconfig` tool[9], which is included in the *Bluez Stack*. An example of output is present in the following image. As reported in the manual, `hciX` is the name of a Bluetooth device/adapter installed in the system. (only one, in this example).

The *check_bt_status()* function works by identifying each adapter through the following code, which match lines that **starts with a string like "hciX"**:

[9] hciconfig manpage



```
sec@XPS:~$ hciconfig
hci0:   Type: BR/EDR  Bus: USB
        BD Address: 9C:B6:D0:16:C2:3E  ACL MTU: 1024:8  SCO MTU: 50:8
        UP RUNNING PSCAN
        RX bytes:8193 acl:72 sco:0 events:541 errors:0
        TX bytes:7318 acl:73 sco:0 commands:385 errors:0
```

Fig. 32: Output of *hciconfig* tool

```
if [[ "$line" =~ ^hci[0-9] ]]; then
    ..
fi
```

For each line identified (i.e. *a BT adapter*), it's possible to understand if it represents a working or a disabled adapter by checking the presence of `UP` or `DOWN` (*on* and *off* respectively) in the following two lines. This is done through the following code:

```
read #skip a line
read adapter_status

if [[ $adapter_status == *"UP"* ]]; then
    #bluetooth is active
else
    #bluetooth is disabled (related to this adapter)
    #save the adapter' status (for log purpose)
fi
```

If an active adapter is detected, it means that Bluetooth *should* be working. The function then terminates *without providing output*. Otherwise a list of detected devices and their status is printed.

---

scan_address()

```
input: <device_name> <file_name> [timeout]
output: MAC address of <device_name> (-1 if not found)
```

It scans for `[timeout]` seconds (15, if no value is provided) looking for a BLE device called `<device_name>` and saves the contents of the advertising packages in `<file_name>`.

After a first check on the presence of the necessary parameters, the following line of code carries out the essential part of the work.

```
scan_results=$(sudo timeout $timeout hcitool lescan --duplicate &
    sudo timeout $timeout hcidump --raw > $2)
```

Note that:

- *Hcitool* starts scanning for BLE devices. The output produced is saved in the `scan_results` variable
- *Hcidump* intercepts the Advertising packages and **redirects everything in a file** identified by the second parameter `$2` (which corresponds to `<file_name>`)
- The `&` operator allows to run *hcitool* in the background (but still producing output) while running *hcidump*
- using `timeout` is convenient for killing both tools, which otherwise need to be stopped manually

At the end of the timeout, a `while` loop scrolls through the *hcitool* output (saved in the variable `scan_results`), looking for the MAC address of the BLE device `<device_name>`. If this is found then the loop is interrupted and the address is returned through an `echo` command. Otherwise the value `-1` is returned.

Note: Calling this function generates a `<file_name>` file. It contains the *raw* output of `hcidump`.

format_address()

```
input: <MAC_address>
output: formatted address
```

It formats the MAC address provided as input to make it conform to the format used by `hcidump`. To do this it is necessary to replace each `:` character with a *white space* and **reverse** the order of each byte (the first becomes the last, the second becomes the penultimate etc..)

As an example, the address `00:11:22:33:44:55` becomes `55 44 33 22 11 00`.

find_weight()

```
input: <MAC_address> <hcidump_file>
output:
    weight in Kg
    (-1 if <hcidump_file> does not contain adv packages sent from the␣
↪scale)
```

It calculates the **weight in Kg**, given the MAC address of the scale and the name of the file previously generated by `hcidump` (and thus obtained with `scan_address`).

The function checks for the presence of the required parameters and the existence of the file. Then it starts analyzing the file *package by package*. This is the example structure of the packages we want to identify (formatted according to the *hcidump* standard):

```
> 04 3E 29 02 01 03 01 D0 FF FF FF FF FF 1D 0F FF 02 A1 09 FF
  02 F8 FF FF 82 FF FF 21 71 AA 02 01 06 09 09 59 6F 48 65 61
  6C 74 68 B6
```

The beginning of a package is marked with the > symbol and in the first line it contains the MAC address of the device that sent it (`D0 FF FF FF FF FF` in the example). This allows us to identify the packages we are potentially interested in. Within these, the information concerning the weight is contained in the **first two bytes of the second row** (`02 F8` in the example).

Based on these considerations, the script reads the file looking for lines containing both the correct MAC address and the > character through this code:

```
while read line
do
    if [[ $line = *'>'* && $line = *"$address"* ]]; then

        ..
        ..
    fi
done < "$hcidump_file"
```

If a consistent line is found, the next one is saved in a variable. However, our goal is to identify the second line of the **last consistent package**, because it is the one containing the most accurate weight. For this reason, the cycle does not break at the first detected occurrence, but continues the inspection *until the end of the file*.

Once the cycle is complete, we have the desired line stored in the `$last_occurrence` variable. If no packet has been detected, the variable contains an *empty string* and **-1** is returned.

Having the desired line available, the function extracts the first two bytes and carries out the operations required to obtain the weight (concatenation, conversion to base 16, division by 10). The result is returned with an `echo` command.

get_weight.sh File

This script exploits the functions previously exposed to automate the reading of the weight from the BLE scale. To do this two variables are defined:

```
hcidump_file="hcidump_output"
available_time=15
```

The first one indicates the name of the *hcidump* file that will be generated by the function *scan_address*. The second allows you to select the time you have available to weigh yourself from the moment the script is started or the password is entered (if required).

The order in which the operations are carried out is as follows:

1. import the functions defined previously:

   ```
   ./basic_functions.sh
   ```

2. check Bluetooth status, scan for the MAC address and save the output in a variabile (it also generates the `hcidump` file)

   ```
   check_bt_status
   MAC_address=$(scan_address YoHealth $hcidump_file $available_time)
   ```

   NOTE: *YoHealth* is the name of our target BLE device

3. if the MAC address has been found, analyze the `hcidump` file:

   ```
   if [[ $MAC_address = "-1" ]]; then
        echo "get_weight.sh error: MAC address not found"
   else
        find_weight $MAC_address $hcidump_file
   fi
   ```

4. remove the `$hcidump_output` file (now useless)

Note: In order to run the scripts you need **execution permissions**:

```
$ cd /path/to/scripts/directory
$ chmod u+x *.sh
```

Then to run the script:

```
$ ./get_weight.sh
```

2.5 Contributions

The writing of this guide has mostly developed around a single BLE device. However, this document *does not* mean to be a "closed chapter". Instead, it would like to be an evolving project, in which to gather information on reverse engineering techniques and to make available works already done in this area.

Every contribution, regarding **any section** of the guide, is welcome. If you are interested, check out our GitLab page.

2.5.1 Add Other BLE Devices

If you reverse-engineered a BLE device (*even just some features*) and you want to add it to this project these are short guidelines to follow. Basically, you should document your work in these two sections:

1. *Protocol Description* - **mandatory**
2. *Script Creation* - if you wrote some scripts/software

"Protocol Description" Section

This section should contain, not necessarily in this order:

- A **description of the BLE device**, its features and how to use it. In addition, you can add details, useful information and issues related to the *product* or *company*. (e.g. permissions required by the Android application or discussions/contacts with the company)
- **Information on Bluetooth communication**. That is: what features of the BLE are used to exchange data between the peripheral and the central device. (e.g. services/characteristics used, useful information in advertising packages, optional pairing)
- A **detailed description of the protocol syntax**. Describe accurately the composition of the commands, of the notifications or in any case of all the data concerning the reverse-engineering protocol.
- *Optional* - If the protocol is composed of several commands/notifications, represent it using a **formal method** for syntax and grammars (e.g. *EBNF*)

"Script Creation" Section

This section is quite free, depending on the scripts/software created and how they work. In general, it should contain:

- **General information** on what the scripts do, the system on which they were tested and *the license* (essential!).
- **Information on the software required** to use the scripts and possibly an installation guide.

  If you think the tools used are interesting, you can add a section about these at the top of the page, after the one on the *Bluez Stack* (or integrating it).

- **A description of:**
  - **how to use the scripts** (e.g. *permissions*, *what each script does*)
  - **interesting aspects of the implementation** of each individual file/function (so that the end user understands how it works)
- *Optional* - Tests, analysis, future developments or other interesting details.

Note: Scripts and other material can be uploaded to a *devicename_reveng* directory on our GitLab repo

2.5.2 Translations

Translations of this guide in other languages (*even just small parts*) are always welcome. If you are interested, contact us!