University of Verona

DEPARTMENT OF COMPUTER SCIENCE
Master Degree in Computer Science and Engineering

Master Degree Thesis

A security analysis of Android Things

Candidate:
Alessandro Cosma
VR420117

Thesis advisor:
Prof. Massimo Merro

Academic Year 2018-2019

Sommario

In questi ultimi anni stiamo assistendo ad una rapida espansione della tecnologia IoT, basata su dispositivi connessi alla rete in grado di acquisire, selezionare e trasmettere dati ed effettuare azioni sulla base delle indicazioni ricevute o in funzione di una capacità elaborativa locale.

Questo rapido sviluppo è accompagnato soprattutto dal crescente interesse per il mondo IoT da parte delle maggiori aziende IT come Google, Apple, Microsoft e Amazon, i quali negli ultimi anni hanno investito parecchie risorse per lanciare nel mercato le proprie piattaforme e i loro dispositivi legati all'Internet of Things. Google, in particolare, ha sviluppato un nuovo sistema operativo chiamato Android Things, derivato da Android ed ottimizzato per i sistemi IoT.

Se da un lato lo sviluppo di queste tecnologie porta molti vantaggi nei suoi campi di utilizzo, quali la domotica, l'automazione, i trasporti, ecc, dall'altro richiede una vasta gamma di competenze, per affrontare le varie problematiche e criticità dovute alla sua continua crescita, come l'interoperabilità tra i dispositivi, la sicurezza delle connessioni utilizzate, l'autenticazione e l'integritá dei dati trasmessi, nonchè i requisiti di privacy come la protezione dei dati e la confidenzialità delle informazioni personali.

In questa tesi proponiamo un'analisi della piattaforma Android Things di Google, offrendo una panoramica delle sue caratteristiche, del suo funzionamento e delle principali tecnologie fino ad ora supportate. Sarà presentato nel dettaglio un approccio per lo sviluppo di codice IoT basato sulle Android Architecture Components, delle librerie per la progettazione di software Android. Sarà infine presentata un'analisi relativa all'utilizzo della tecnologia Bluetooth Low Energy in Android Things, in cui evidenzieremo alcuni imporanti problemi legati alla sicurezza della connessione.

Abstract

In these last years, we are seeing a rapid expansion of the IoT technology, based on devices connected to the network and able to acquire, select and transmit data and carry out actions based on the indications received or according to a local processing capacity.

This rapid growth is due to the increasing interest in the IoT world by major IT companies, like Google, Apple, Microsoft and Amazon, which in recent years have invested a lot of resources to launch their devices and their platform on the Internet of Things market.

If on the one hand the development of these technologies brings many benefits in its different areas of use, such as domotics, automation, transport, etc., on the other hand it requires a wide range of skills, to address the various critical issues due to its continuous growth, such as the interoperability between the devices, the security of the connections used, the authentication and the integrity of the transmitted data, as well as the privacy requirements like data protection and the confidentiality of personal information.

In this thesis we propose an analysis of the Android Things platform, offering an overview of its features and the main technologies supported. We will present a detailed approach to the development of the IoT code based on Android Architecture Components, a set of libraries to design the Android software. Finally, we will present an analysis of the use of Bluetooth Low Energy technology in Android Things, where we will highlight some important issues related to the security of the connections.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications. The basic idea of this concept is the pervasive presence around us of a variety of things or objects, such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc, which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals[1].

Nowdays IoT devices are widespread in everyday-life and have a high impact on behavior of potential users. They find usage in private field like domotics, assisted living, e-health, enhanced learning, but also in the business area, for example automation and industrial manufacturing, logistics, business/process management, intelligent transportation of people and goods.

In 2009 the number of the connected IoT devices had surpassed the number of people living on this planet and, like Gartner projects [7], in 2020 it is estimated that there will be 20.4 billion IoT devices.

Although IoT development is rapidly increasing and on the one hand it brings many advantages, on the other hand it needs complex technologies that require a wide range of expertise. Many issues need to be taken into account.

The major issue concerns the *interoperability* of the devices: technologies developed by different manufacturers, with different characteristics and different ways to operate have to interact with each other. Moreover, this full interoperability of the devices must provide them a high degree of smartness by enabling their adaptation and autonomous behavior, while guaranteeing trust, privacy, and security.

Regarding the increase of the IoT devices there are *addressing and networking issues*: the high number of connected nodes, each of which will produce sharable information, requires addressing policies. In the IPv4 protocol, the number of available IPv4 addresses is decreasing rapidly and will

soon reach zero, so it is clear that other addressing policies should be used. The IPv6 addressing has been proposed for some of new IoT technologies (e.g. 6LOWPAN).

Another important issue regards the *security*. Most of the IoT devices spend their time unattended, so a physical attack is easy to perform but, a more critical aspect concerns the communications between these components: most of the connections are wireless, which makes eavesdropping extremely simple. In the end, IoT components are characterized by low energy capabilities and limited computing resources, so they cannot implement complex schemes supporting security.
More specifically, the major problems related to security concern authentication and data integrity. Authentication is difficult as it usually requires appropriate authentication infrastructures and servers that achieve their goal through the exchange of appropriate messages with other nodes[2]. Data integrity, even if is an aspect widely studied in the traditional computing, it is subject to new problems because the IoT devices spend most of the time unattended and data can be modified by adversaries even when they are stored in the node as well as they cross the network.

Finally, another important challenge regards the *privacy requirement*: data protection and confidentiality of the user's personal information have to be ensured, since devices may manage sensitive information.

During the last few years, new IoT platforms have been introduced by the most important IT players: Google has presented the Android Things OS and Google Home, Apple launched the HomeKit framework, Microsoft released Windows IoT and Azure Sphere OS and Amazon presented Greengrass, a software that extends the functionality of the cloud to the IoT local devices. Each of these IT companies has developed its own set of IoT APIs, in order to integrate the various protocols of the IoT world in the development of its software and thus in its apps.

While on the one hand, it is important to guarantee security in IoT communication protocols, on the other, there is the need to guarantee the security of the APIs that use these protocols, understanding how they interact with each other and with the operating system running on the devices.

Contribution

In this thesis, we will analyze the Android Things OS, an operating system developed by Google to build IoT projects, which is based on Android OS. With this operating system, Google wanted to create a stripped-down version of Android aimed at on cheap and low-power IoT hardware and add this OS with the Android family alongside Android TV, Android Auto, and Android Wear.

Our analysis can be defined as a "security analysis" because, after presenting a general overview of the Android Things OS, with its peculiarities, and the differences with Android OS we will focus on the correct use of its IoT features. Then we will propose a better code development, which makes use of Android Architecture Components and, unlike from the code developed up to now and available online, it respects the life cycle of the application's activities, increases the modularity of the application and makes code maintenance and its readability easier. We will analyze our developed code with Julia, a static analyzer for Java and Android.
In the end, we will focus on the use of some wireless communication APIs (Bluetooth Low Energy in our case) and will report some important bugs and vulnerabilities that we have detected.

Outline

Chapter 2 lays out an overview of the Android Things OS where we present its hardware and software characteristics and peculiarities, its differences with Android and its IoT features concerning the interaction and the management of GPIO, PWM and I2C peripherals.

Chapter 3 discusses the use of the Android Architecture Components as a better programming approach when the Android Things code is developed. This chapter presents this components and exposes a comparison of an application with and without their use.

Chapter 4 describes the use of Bluetooth Low Energy API for Android Things. There is a brief description of BLE technology and after that it is presented the BLE API for the Android Things OS, through the description of two projects that connect an Android Things device and an Android device. In the end, we present some relevant bugs in the Bluetooth LE API, which we identified during the development of the two projects.

We end with Chapter 5 as a conclusion summarizing our results and some possible future research ideas.

Chapter 2

Android Things

Android Things is an operating system developed by Google to build IoT projects. It was announced on December 13^{th}, 2016 and released in its commercial version in May, 2018 with the 1.0 release.

Android Things is not the first step for Google in IoT world. During the I/O 2015 developer conference Google introduced Brillo and Weave as components of its new Internet of Things platform. The aim of Brillo was to bring simplicity and speed of software development to hardware by joining together three major components: embedded OS based on Android (Brillo operating system), core service for enable a great getting-started experience and to allow to operate at a scale and a developer kit with tools to build, test and debug the applications. Brillo should have allowed hardware developers to quickly prototype and develop compatible devices with its embedded operating system, while Weave was a communication platform which provide the comunication between devices and other apps. Weave should also have handled user setup.
At the end Brillo was never officially released and remained in developer preview mode, that was available on Google's developers site.

In December 2016, we have witnessed to a reboot of Google's Android IoT strategy. It announced Android Things like successor to Brillo and released the first SDK preview. Android Things, compared to Brillo, belongs to a new IoT view, because reinstates the full Android enviroment. It allows development to be accomplished with the same developer tools as standard Android, whereas Brillo didn't offer that. Brillo didn't catch on because developers likely found it difficult to jump in and work on a new product.
Now, because they have access to the same familiar tools, they can quickly get up to speed. Android Things indeed, makes use of tools like Android Studio, Android SDK, Google Play Services, and Google Cloud Platform. This platform uses Google's Weave communication protocol, which is also getting an update to add compatibility for Google Assistant. Philips Hue and Samsung SmartThings already work with Weave. Android's developers

can reuse their Android knowledge to implement smart Internet of Things projects; so thanks to this great potential they can smoothly move to IoT and start developing and building projects quickly.

Since its release Android Things gets more than ten minor update and currently it is available in its official stable platform releases 1.0.11 (April, 2019).

2.1 Overview

Android Things is an operating system based on Android, simplified and polished to run smoothly on low-power IoT devices. Though Android is part of the name of this OS, one should not think of Android Things as an OS that would run alongside an Android phone or an Android Wear device. In fact Android Things is an OS that works in the background in fully transparent way. It allows smart devices to handle their own tasks rather than let servers do the processing. Because it is capable of more complex tasks, it's ideal for complex smart devices like printers and locks, rather than basic power outlets.

It is well known that Android OS makes use of Linux kernel at its core and Linux is a full multi-tasking operating system with virtual memory support. Also Android Things needs a processor that guarantees and supports the virtualization of memory, therefore a processor with a full MMU. For ARM processors this means something from the Cortex-A range and not a microcontroller from the Cortex-M range.

It is important to underline that there are many IoT products that use microcontrollers and therefore they need less memory, less power, and use less complex operating systems. The choice of using Android, and especially Linux, has addressed Google in a particular segment of the IoT market.

2.1.1 Android Things hardware

Android Things enables programmers to develop applications on popular hardware platforms. To date, the System on Module (SoM) supported by Android Things are NXP Pico i.MX7D and Raspberry Pi 3 Model B [8].
The Board Support Package (BSP) is managed by Google, so it is not required kernel or firmware development. Software images are built and delivered to devices through the Android Things Console. This gives a trusted platform to develop on with standard updates and fixes from Google.

NXP i.MX7D Starter Kit

To build and test the applications that will be proposed in the following chapters of this work, we made use of an hardware platform called NXP i.MX7D Starter Kit [9]. This platform is based on NXP Pico i.MX7D and

Figure 2.1: NXP Pico i.MX7D Starter Kit

in addition to the SoM, it has useful modules that simulate the sensors that may be available on a real IoT device. All the modules are fully integrated with the Android Things OS thanks to their dedicated libraries, that we will present in Section 2.3.

By default, the latest version of Android Things is installed on the i.MX7D development board. Otherwise, it is possible to flash the Android Things image onto the hardware by downloading the Android Things Setup Utility from the Android Things Console, and through this utility follow the installation wizard.

The Figure 2.1 shows the components of to the kit, which are:

- **Pico i.MX7Dual** board computer: it features an advanced implementation of two ARM®Cortex®-A7 cores and delivers high-performance processing for low-power requirements with a high degree of functional integration.
- **Rainbow HAT**: is a board with a buffet of sensors, inputs, and displays to explore Android Things.
- **Camera module**: it is connected to the SOM and it allows to take pictures and record videos.
- **Multi-touch display**: a 5-inch widescreen LCD with LED backlight.
- **Wifi antenna and extender cable**.

The core components of this kit are Pico i.MX7Dual and Rainbow HAT.

Pico i.MX7Dual board

The PICO-PI-IMX7 is a 2-board development board consisting of a System-on-Module and a carrier baseboard and optimized for the Internet of Things. The i.MX7 is an ultra-efficient processor family with featuring NXP's advanced implementation of the ARM Cortex®-A7+M4 core, which operates at speeds of up to 1 Ghz.

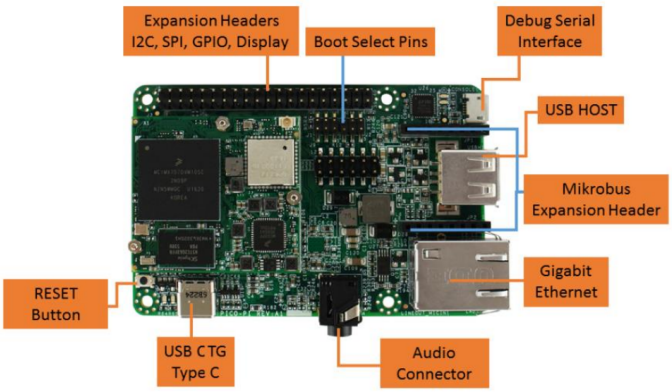


Figure 2.2: PICO-PI-IMX7 Connector Overview

The PICO-IMX7 System-on-Module (PICO-IMX7-EMMC) has 3 Hirose high-speed 70-pin board-to-board connectors and integrates the NXP i.MX7, Memory, eMMC, Power Management IC (PMIC) and WiFi / Bluetooth on the module.

The PICO-PI-IMX7 Carrier Baseboard (PICO-PI-GL) has 3 Hirose high-speed 70-pin board-to-board connectors that connect to the System-on-Module and provides the real-world interfaces such as audio, network, USB and a large number of signals on the various pin headers [3].

The PICO-PI-IMX7 has a number of expansion headers that can be used to connect sensors, motors, and external devices, for example the Rainbow HAT for Android Things.

Rainbow HAT

Rainbow HAT is a board developed by Pimoroni in collaboration with the Android Things team to create this add-on board that features displays, sensors, sound, and LEDs. Rainbow HAT communicates with Pico i.MX7Dual through the 40-pin connector presents on it (see Figure 2.2) and has a lot of features crammed into it, in particular it has:

- seven APA102 multicolour LEDs;
- four 14-segment alphanumeric displays (green LEDs);
- HT16K33 display driver chip;
- three capacitive touch buttons: A, B and C;
- Atmel QT1070 capacitive touch driver chip;

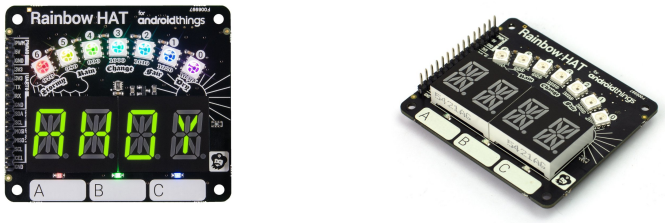


Figure 2.3: Rainbow HAT

- blue, green and red LEDs;
- BMP280 temperature and pressure sensor;
- Piezo buzzer;
- breakout pins for servo, I2C, SPI, and UART (all 3v3);

Rainbow HAT is designed specifically to show off the wide range of protocols available on the Pico i.MX7Dual , including SPI (the APA102 LEDs), I2C (the BMP280 sensor and 14-segment displays), GPIO (the capacitive touch buttons and LEDs), and PWM (the piezo buzzer).

2.1.2 Android Things OS

The Android Things OS has a structure much more compact than Android: the applications for Android Things have less layers beneath and this means that are nearest to drivers and peripherals than normal Android apps.

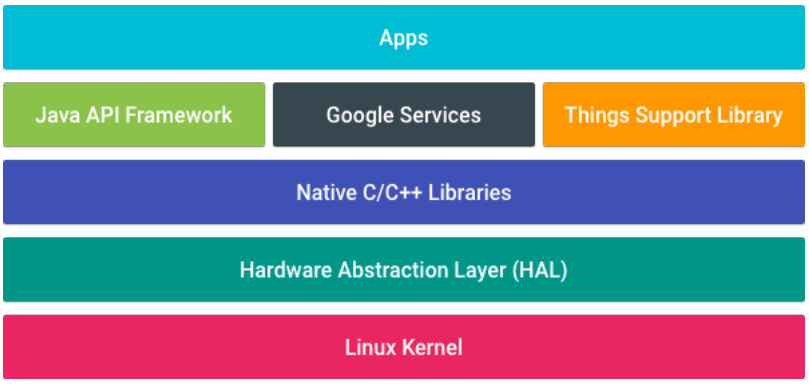
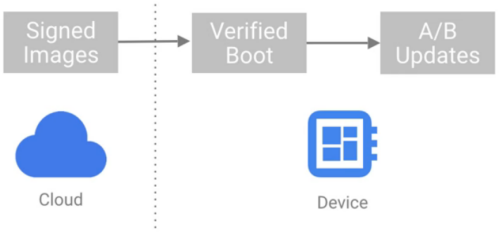


Figure 2.4: Android Things OS layer structure

To improve the Android experience for embedded use, the platform has been turned for faster boot times and lower memory footprint, including a smaller variant of Google Play Services optimized specifically for IoT, the

Things Support Library. Android Things provides also new APIs through the Android Things Support Library, to better integrate apps with costum hardware: this includes expanded support for peripheral interfaces and device management. Android Things does not include user-facing apps (launcher or browser): it is designed to boot directly into the apps built for the device.

Secure Device Updates



Google takes care of providing updates and security patches to the core of Operating System. System images are signed by Google and verified for integrity on devices. This is a prevention against corrupt or tampered update from being applied. Android's seamless A/B updates [1] apply the image in the background without interrupting the app. Failure to boot, following an update, will roll back to the previous version, ensuring the system always boots into a known good state.

Android Things Console

The Android Things Console provides tools to install and update the system image on supported hardware devices. Using the console developers can download and install the latest Android Things system image, build factory images that contain OEM[2] applications along with the system image and also push Over-the-Air (OTA) system image updates to devices.
The Android Things Console also provides easy and secure deployment of updates for the applications installed on the connected devices. Through the console it is possible to monitor informative analytics to understand how well products are performing. All the technology and the infrastructure to host and deliver updates are managed by Google.

[1] A/B system updates, also known as seamless updates, ensure a workable booting system remains on the disk during an over-the-air (OTA) update.

[2] An original equipment manufacturer (OEM) is a company that produces parts and equipment that may be marketed by another manufacturer.

2.1.3 Things Support Library

Things support library is the new library developed by Google to handle the communication with peripherals and drivers. This is a completely new library not present in the Android SDK and this library is one of the most important features. It exposes a set of Java Interface and classes (APIs) that we can use to connect and exchange data with external devices such as sensors, actuators, etc[4].

This library exposes **Peripheral I/O APIs** that let applications to comunicate with sensor and actuators, abstracting the details of internal communication. The library supports several industry standard protocols such as:

- **GPIO**: General-Purpose Input/Output.
- **I2C**: Inter-Integrated Circuit.
- **PWM**: Pulse-Width Modulation.
- **SPI**: Serial Peripheral Interface.
- **UART**: Universal Asynchronous Receiver-Transmitter.

In addition, in this library, programmers can find **User Driver APIs**: the aim of this set of APIs is the creation and the registration of new device drivers called *user drivers*. We will discuss the User Drivers in Section 2.3.4.

Another important feature introduced with Android Things is the enhancement of the **Android Bluetooth APIs**. This improvement concerns the device state management and allows to configure active Bluetooth profile, device attributes and capabilities. Furthermore, it is possible to control the connection process, initiate pairing with a remote device and handle incoming pairing requests.

Android Things also supports the connection between devices through IP-based Low-Power Wireless Personal Area Networks (LoWPAN). New **LoWPAN APIs** have been introduced into Things Support Library to enables developers to manage and configure IP-based, low-power, lossy networks. These APIs enable applications to perform a scan for nearby supported networks, to join a specific network and to form new ones.

Into Things Support Library new **TimeManager APIs** have been introduced to provide methods to check the time and time zone of the clock, date/time formats and automatic time synchronization with the network.

2.2 Platform differences between Android and Android Things

In this section we are going to describe the main differences between Android Things and Android. Due to the fact that Android Things is optimized for embedded devices, there are some features and APIs of Android that are not supported by it.

Unsupported APIs

Since Android Things is addressed for IoT world, which is characterized by devices working without user interactions, Graphical User Interfaces (GUI) are optional because not all devices include a display. However, the presence of GUI strictly depends on the type of application which is developed, and sometimes the presence of a display is necessary. The advantage of developing using Android APIs is that the UI process is the same as in Android. Taking advantage of this feature programmers can develop an IoT UI easily and fast, re-using Android knowledge.

Table 2.1 outlines the Android features and the affected framework APIs which are not supported by Android Things.

Feature	API
System UI	NotificationManager
	KeyguardManager
	WallpaperManager
VoiceInteractionService	SpeechRecognizer
android.hardware.fingerprint	FingerprintManager
android.hardware.nfc	NfcManager
android.hardware.telephony	SmsManager
	TelephonyManager
android.hardware.usb.accessory	UsbAccessory
android.hardware.wifi.aware	WifiAwareManager
android.software.app_widgets	AppWidgetManager
android.software.autofill	AutofillManager
android.software.backup	BackupManager
android.software.companion_device_setup	CompanionDeviceManager
android.software.picture_in_picture	Activity Picture-in-picture
android.software.print	PrintManager
android.software.sip	SipManager

Table 2.1: Unsupported features and APIs

Content providers

Another important difference from Android concerns *content providers*. Content providers are a mechanism that helps an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process [12].
Android Things does not include the standard suite of system apps and content providers. Android Things official documentation suggests avoiding the use of the following content provider APIs in applications:

CalendarContract	ContactsContract	DocumentsContract
DownloadManager	MediaStore	Settings
Telephony	UserDictionary	VoicemailContract

Table 2.2: Deprecated content providers APIs

Runtime permissions

In an Android enviroment, running applications "live" in a limited-access sandbox. When an application needs to access a specific resource outside the sandbox it has to request permission. Usually, the declaration of needed permission is in Androd Manifest File which describes essential information about the application, including the permissions. Afterward, when the user launches the application, each permission has to be approved at runtime (on Android 6.0 and higher).

In Android Things the granting of app permissions is done differently. As mentioned above, lots of IoT applications do not require a user interface or input device, so the granting of permissions at runtime is not always feasible; for that reason all the permissions in Android Things are assigned using Android Studio or the Android Things Console.
When an app is run from Android Studio, all permissions (including dangerous permissions [3]) are granted at install time. Developers can use the adb tool to grant or revoke permissions for testing. When an application is ready to be distributed through the Android Things Console, the developer grants the dangerous permissions (instead of the end user) for the application as part of the build creation process. Permissions can be overridden during development, but not on actual products; end users cannot modify these permissions.

[3]Dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps. For example location data (`ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`)

Google APIs

Android Things is compatible with a limited number of Google APIs. As a rule, APIs that require user input or authentication credentials are not available to apps.
Table 2.3 outlines the available and not available Google APIs:

Supported APIs	Unavailable APIs
Awareness	Google AdMob
Cast	Google Pay
Google Analytics for Firebase	Drive
Firebase Authentication	Firebase App Indexing
Firebase Cloud Messaging (FCM)	Firebase Dynamic Links
Firebase Crashlytics	Firebase Invites
Firebase Realtime Database	Firebase Notifications
Firebase Remote Config	Play Games
Firebase Storage	Sign-In
Cloud Firestore	
Fit	
Instance ID	
Location	
Maps	
Nearby Connections	
Nearby Messages	
Places	
Mobile Vision	
SafetyNet	

Table 2.3: Google APIs for Android Things

Another important aspect related to Google APIs is the management of Google Play Services.
The Google Play Services client library is a library that Google needs to add or update Android features on devices, regardless of the Android version and the manufacturer. Since Android is very fragmented, in terms of distribution of system updates, Google has decided to introduce this app to "check" its functions and have them available to all devices even if they have different software.
Each release of Android Things bundles the latest stable version of Google Play Services, and requires at least version 11.0.0 of the client SDK. It is important to stress that Android Things does not include the Google Play Store, which is in charge of the automatically update of Play Services on the device, so due to the static nature of the Play Services version on the device, applications cannot target an SDK greater than the version bundled with the current release.

ANR - Application Not Responding

In the Android OS, when the UI thread of an app is blocked for too long, an "Application Not Responding" (ANR) error is triggered. If the app is in the foreground, the system displays a dialog to the user. The ANR dialog gives the user the opportunity to force quit the app[13].

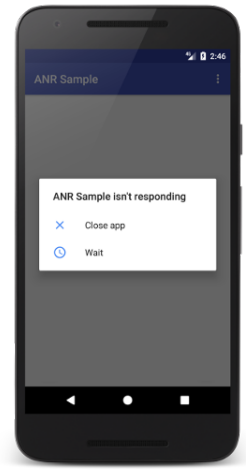


Figure 2.5: ANR dialog displayed to the user

Android displayes the ANR dialog when it detects one of the following conditions:

- No response to an input event (such as key press or screen touch events) within 5 seconds.
- A BroadcastReceiver has not finished executing within 10 seconds.

During the development of some Android Things apps and testing their behavior, we noticed that the ANR implemented in the Android Things OS acts differently.
In this case, when the ANR error is triggered, no dialog are given to the user (even if there is a display in the Android Things device) and there is no opportunity to choose whether to force quit the app or to wait the end of the blocking process. Therefore, an ANR event in an Android Things application kills its execution.

This difference in the behavior of the ANR, is due to the fact that Android Things is designed for use in which the user does not interact directly with the device.

It is important to keep in mind this behavior, in order to designed correctly the implementation of long blocking operations in the code.

2.3 IoT features

The power of Android Things is emphasized when an application is connected and communicates directly with hardware peripherals. In this section will be presented the Peripheral I/O APIs, which enable apps to use industry standard protocols and interfaces to connect to hardware peripherals; in particular we will expose GPIO, PWM and I2C APIs.
Then, we will focus on User-space Driver APIs which allow developers to inject new hardware into the Android framework and make interfacing with peripherals easier.

2.3.1 General Purpose Input/Output (GPIO)

General-purpose input/output (GPIO) pins are physical pins on an integrated circuit that can be controlled via software. These pins provide a programmable interface for the reading of binary input device state, (e.g. pushbutton switch) or the controlling of the on/off state of a binary output device (e.g. LED). GPIO pin only understands logic values (TRUE/FALSE) which maps physically to a low value or high value.

First of all, when a GPIO is used it is necessary to add the required permission for the API to the manifest file. This rule is also necessary for PWM and I2C.

The corresponding Android permission is `USE_PERIPHERAL_IO`:

```
<uses-permission android:name="com.google.android.things.permission.USE_PERIPHERAL_IO"/>
```

In order to open a connection to a GPIO port, it is necessary to know the unique port name. Every GPIO port in a board, for example in NXP Pico i.MX7D, has a unique identifier, called BCM (Broadcom pin number). Using the method `getGpioList()`, belongs to `PeripheralManager` [4] class, it is possible to discover all the available port names.

```
PeripheralManager manager = PeripheralManager.getInstance();
List<String> portList = manager.getGpioList();
if (portList.isEmpty()) {
    Log.i(TAG, "No GPIO port available on this device.");
} else {
    Log.i(TAG, "List of available ports: " + portList);
}
```

[4] `PeripheralManager` is the system service responsible for managing peripheral connections.

Once the target name is known, using `openGpio()` it is possible the connection to that port. When developing with this type of components, it is important to keep in mind that the new connection to a port cannot be opened until the existing one is closed. Therefore it is necessary to close the connection when the communication with the GPIO port is done, in order to free up resources. To close the connection, there is the port `close()` method.

```
// The chosen name
private static final String GPIO_NAME = "BMC21"
// Declare the GPIO variable
private Gpio mGpio;

try {
    PeripheralManager manager = PeripheralManager.getInstance();
    //open GPIO pin
    mGpio = manager.openGpio(GPIO_NAME);
} catch (IOException e) {
    Log.w(TAG, "Unable to access GPIO", e);
}
```

Note that the connection management needs to be wrapped in a try/catch block to handle the possibility of a thrown an `IOException`.

Reading from an input

GPIO pins can be used for input or output. Whether the pin is used to read in information, it is necessary to set `DIRECTION_IN` as the pin direction, and also configure the trigger type for the pin; in this way, it knows when to notify the application that something has occurred.

The general steps to follow are:

1. Configure `DIRECTION_IN` mode with `setDirection()` method.
2. Configure the pin voltage represented by an input value of true (active) with `setActiveType()` [optional].

   Two ActiveType constants are available:

   - `ACTIVE_HIGH`: High voltage is active.
   - `ACTIVE_LOW`: Low voltage is active.
3. Access the current state with the `getValue()` method.

Regarding the listening for input state changes, a programmer additionally has to:

1. Attach a GpioCallback to the active port connection.

2. Declare the state changes that trigger an interrupt event. This is done by using the `setEdgeTriggerType()` method.

   The edge trigger supports the following four types:

   - `EDGE_NONE`: No interrupt events. This is the default value;
   - `EDGE_RISING`: Interrupt on a value transition from false to true;
   - `EDGE_FALLING`: Interrupt on a value transition from true to false;
   - `EDGE_BOTH`: Interrupt on all transitions;

3. Return true from within onGpioEdge() to indicate that the listener should continue receiving events for each port state change.

4. In the end, it is important to unregister any interrupt handlers, when the application is no longer listening for incoming events. This is done through `unregisterGpioCallback()`.

Following example synthetizes the reading and the listening from an input:

```
public void configureInput(Gpio gpio) throws IOException {
    // Initialize the pin as an input
    gpio.setDirection(Gpio.DIRECTION_IN);
    // High voltage is considered active
    gpio.setActiveType(Gpio.ACTIVE_HIGH);

    // Register for all state changes
    gpio.setEdgeTriggerType(Gpio.EDGE_BOTH);
    gpio.registerGpioCallback(gpioCallback);
}

private GpioCallback gpioCallback = new GpioCallback() {
    @Override
    public boolean onGpioEdge(Gpio gpio) {
        // Read the active low pin state
        if (gpio.getValue()) {
            // Pin is HIGH
        } else {
            // Pin is LOW
        }

        // Continue listening for more interrupts
        return true;
    }

    @Override
    public void onGpioError(Gpio gpio, int error) {
        Log.w(TAG, gpio + ": Error event " + error);
    }
};
```

Writing to an output

If a programmer wants to set GPIO pin with the purpose of writing information, he has to set the direction out. In this case there are two possibilities: `DIRECTION_OUT_INITIALLY_LOW` or `DIRECTION_OUT_INITIALLY_HIGH`, depending on whether it is necessary the component to start as on or off. No trigger types are required for an output pin.
Similarly to the reading case, the steps to follow are:

1. Configure the pin as an output, using `setDirection()` method with `DIRECTION_OUT_INITIALLY_LOW` or `DIRECTION_OUT_INITIALLY_HIGH`.
2. Configure the pin voltage represented by an input value of true (active) through `setActiveType()`, with `ACTIVE_HIGH` or `ACTIVE_LOW` constant. [optional].
3. Set the current state with the `setValue()` method.

The following code shows how to set up an output pin to initially be high, then toggle its state:

```
public void configureOutput(Gpio gpio) throws IOException {
    // Initialize the pin as a high output
    gpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_HIGH);
    // Low voltage is considered active
    gpio.setActiveType(Gpio.ACTIVE_LOW);
    ...
    // Toggle the value to be LOW
    gpio.setValue(true);
}
```

2.3.2 Pulse Width Modulation (PWM)

Pulse Width Modulation is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on, versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, that pulse width is changed or modulated [14].
There are three important factors in this type of modulation:

- **Frequency**: describes how often the output pulse repeats.
- **Duty Cycle**: represents the width of a pulse within a set frequency. Duty cycle is expressed as a percentage of high signals within a frequency.

- **Period**: represents the time it takes for each up/down cycle to occur. Period is inversely related to frequency.

In order to use PWM API, it is necessary to declare the required permission for it, using the `USE_PERIPHERAL_IO` Android permission.

The connection management is very similar to the previous case: it is possible to retrieve a list of available PWM ports by creating a `PeripheralManager` Service and then calling `getPwmList()` method.
Once the name of the PWM pin is established, the opening of the connection to the peripheral is made by calling `openPwm()` method. This will need to be wrapped in a try/catch block to handle the possibility of a thrown IOException.

The PWM signal activation/deactivation is done through `setEnabled(boolean value)` method: boolean parameter setted to `true` means activated; boolean parameter setted to `false` means deactivated.
There is also the possibility to control setting on PWM pin, such as the frequency and duty cycle, with `setPwmFrequencyHz()` and `setPwmDutyCycle()` methods.

As with GPIO, a new connection to the port cannot be open until the existing connection is closed. Method `close()` stops the connection.

An additional aspect to consider is that the PWM pin continues to output its signal even after the calling of `close()` method. To stop the signal the programmer has to call `setEnabled(false)`.

Following example summarizes the use of PWM API.

```
public class HomeActivity extends Activity {

    private static final String PWM_NAME = ...; // PWM Name
    private Pwm pwm;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Attempt to access the PWM port
        try {
            PeripheralManager manager = PeripheralManager.getInstance();
            pwm = manager.openPwm(PWM_NAME);
            // Set frequency and ducty-cycle
            pwm.setPwmFrequencyHz(120);
            pwm.setPwmDutyCycle(25);
            // Enable the PWM signal
            pwm.setEnabled(true);
        } catch (IOException e) {
            Log.w(TAG, "Unable to access PWM", e);
        }
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();

        if (pwm != null) {
            try {
                // Disable the PWM signal
                pwm.setEnabled(false);
                // Close the connection
                pwm.close();
                pwm = null;
            } catch (IOException e) {
                Log.w(TAG, "Unable to close PWM", e);
            }
        }
    }
}
```

2.3.3 Inter-Integrated Circuit (I2C)

I2C is a synchronous serial interface, which means it relies on a shared clock signal to synchronize data transfer between devices. The device in control of triggering the clock signal is known as the **master**. All other connected peripherals are known as **slaves**. Each device is connected to the same set of data signals to form a bus [15].
Common use cases for I2C are sensors and actuators, for example LCD displays, thermometers and accelerometers.

Through I2C bus, programmers can communicate with multiple devices exploiting a single physical connection. I2C devices requires three pin, different from GPIO and PWM which require only one connection.
The 3-Wire interface consists of:

- **Shared clock signal (SCL)**: delivers the clock signal from the master device to the slave devices.
- **Shared data line (SDA)**: uses for data transmission.
- **Common ground reference (GND)**: An electrical ground connection.

Notice that data is transferred over one wire, that means only *half-duplex* [5] standard is avaiable over I2C. The communication is initiated by the master device and the slave must wait until the transmission from the master has been completed; then it can respond.

Device addressing is very important in I2C connection, since the standard supports multiple slave devices connected along the same bus. Each device

[5] In a half-duplex system, both parties can communicate with each other, but not simultaneously; the communication is one direction at a time.

is programmed with a unique address and only responds to transmissions the master sends to that address. Every slave device must have an address, even if the bus contains only a single slave [15]. The address is seven bit wide and allows 128 different addresses. [6]

In order to manage an I2c connection it is necessary to declare the `USE_PERIPHERAL_IO` permission in the Android Manifest File (likewise GPIO and PWM). After that, it is essential the knowledge of I2C bus name and slave address. The `getI2cBusList()` method in `PeripheralManager` API allows to get the list of I2C buses available. Once the target bus name and the slave identifier are known (the slave identifier is an address which can be found in the peripheral component's datasheet), `openI2cDevice(DEVICE_NAME, I2C_ADDRESS)` method permits to open a connection with a target device. When the application is ready to close on device, it necessary to unregister the I2C device and close the connection with `close()` method.

```
public class HomeActivity extends Activity {
    // I2C Bus Name
    private static final String DEVICE_NAME = ...;
    // I2C Slave Address
    private static final int I2C_ADDRESS = ...;
    private I2cDevice mDevice;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Attempt to access the I2C device
        try {
            PeripheralManager manager = PeripheralManager.getInstance();
            mDevice = manager.openI2cDevice(DEVICE_NAME, I2C_ADDRESS);
        } catch (IOException e) {
            Log.w(TAG, "Unable to access I2C device", e);
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();

        if (mDevice != null) {
            try {
                mDevice.close();
                mDevice = null;
            } catch (IOException e) {
                Log.w(TAG, "Unable to close I2C device", e);
            }
        }
    }
}
```

[6] To prevent address clashes Philips Semiconductors has introduced a 10 bit address scheme. Devices with 7-bit and 10-bit addresses can be connected to the same I2C-bus.

The `DEVICE_NAME` represents the I2C bus (a device can have one or more I2C bus embedded); `I2C_ADDRESS` represents the individual slave on that bus. Therefore, an I2C device is a connection to a specific slave device on the corresponding I2C bus.

Interaction with registers

The comunication with I2C devices can adopt the System Management Bus (SMBus) protocol, in order to place data inside of registers or to retrieve saved data from registers on peripheral device. There are two types of registers [15]:

- **Readable registers**: Contains data the slave wants to report to the master, such as sensor values or status flags.
- **Writable registers**: Contains configuration data that the master can control.

Android Things allows programmers to read or write individual bytes from a register or groups of bytes from multiple registers. Peripheral I/O API for I2C provides the following types of SMBus commands:

- **readRegByte()** and **writeRegByte()**: Read or write a single byte from a specific register.
- **readRegWord()** and **writeRegWord()**: Read or write bytes from two sequential registers on a peripheral in little-endian format.Read or write two consecutive register values as a 16-bit little-endian word
- **readRegBuffer()** and **writeRegBuffer()**: Read or write bytes from a maximum of 32 consecutive registers. Values are written or retrieved as a byte array.

```
// Modify the contents of a single register
public void setRegisterFlag(I2cDevice device, int address) throws IOException {
    // Read one register from slave
    byte value = device.readRegByte(address);
    value |= 0x40;
    // Write the updated value back to slave
    device.writeRegByte(address, value);
}
// Read a register block
public byte[] readCalibration(I2cDevice device, int startAddr) throws IOException {
    // Read three consecutive register values
    byte[] data = new byte[3];
    device.readRegBuffer(startAddr, data, data.length);
    return data;
```

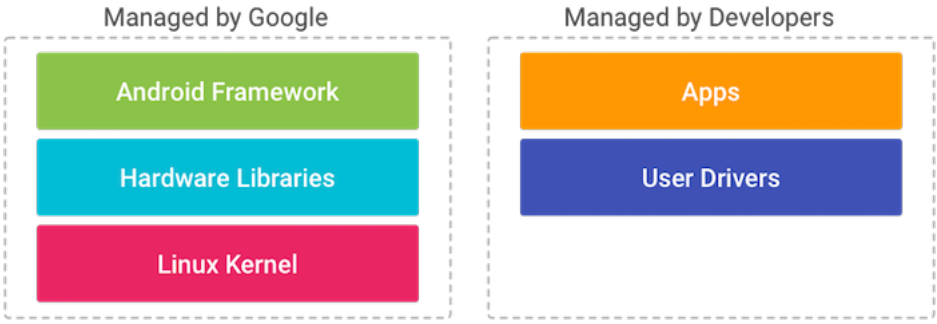
Raw data transfer

Peripheral I/O API also defines raw methods for the full control of bytes transmitted across the wire. These methods, `read()` and `write()`, execute a single I2C transaction. It is important to stress that I2C API does not define an explicit maximum length of data for a raw transaction, but the I2C controller hardware on the device may have a limit number of bytes it can process. It is essential to consult the device hardware documentation in case of large data transfers.

2.3.4 User Drivers for Android Things

The concept of *user driver* has been introduced in Android Things to allow app developers to register new device drivers with the framework. A user driver is a component registered from within apps that extend existing Android framework services through the injection of hardware events that other apps can process using the standard Android API

The benefit of introducing user drivers is a better code portability, due to the application code can be executed on different boards without additional abstractions for the device driver implementation. Programmers can also reuse existing fragment of code and libraries without modify them, to handle the various hardware implementation. User drivers contribute to the better aggregation of data from different services, which Android usually collects together in order to enhance the information reported to apps.



Android Things defines four different types of device drivers:

- **Location**: allows applications to post updates to the physical location of the device using the Android location services. The API supports GNSS and GPS.

- **Human Interface Devices (HID)**: provide a binding between human interaction and the Android Input pipeline, so allow devices to properly react to user action.

- **Sensors**: extend the Android sensor framework with additional sensor devices, connected over the Peripheral I/O.
  The Android sensor framework implements *sensor fusion* to combine the raw data from multiple physical sensors into a single virtual sensor (e.g. accelerometers and gyroscopes). The connnection of a sensor to the framework, using a user driver allows the data it produces to be included in sensor fusion [16].

- **LoWPAN**: with these drivers programmers can extends LoWPAN framework by adding new LoWPAN interfaces connected over Peripheral I/O, for example integrate an external radio module into the LoWPAN API.

Rainbow HAT drivers

In Section 2.1.1 we have introduced NXP i.MX7D Starter Kit as the hardware platform which has been used during the various tests executed with Android Things OS.
Since the various sensors in the kit are supplied by Rainbow HAT, we will now focus on user drivers developed for the Rainbow HAT board and for its releated sensors.

On GitHub there is a growing library of open source user drivers for Android Things named **contrib-drivers** [7]. As remarked in the official GitHub page, the drivers are not production-ready and they are offered as sample implementations of Android Things user space drivers for common peripherals, as part of the Developer Preview release. This means there is no guarantee of correctness, completeness or robustness.

Following table summarizes all avaiable sensors in Rainbow HAT and their relative standard protocol.

GPIO	PWM	I2C	SPI
Capacitive buttons	Piezo Buzzer	BMP280[8]	APA102 RGB LEDs
LEDs	Servo header	HT16K33[9]	

Table 2.4: Available sensors on Rainbow HAT

[7] This library is available at the following repo: `https://github.com/androidthings/contrib-drivers`.

[8] A temperature and pressure sensor.

[9] A segment display.

In order to use the Rainbow HAT driver it is only necessary to add Gradle dependency to the application-level Gradle file .

```
dependencies {
    compile 'com.google.android.things.contrib:driver-rainbowhat:<version>'
}
```

where `<version>` is the last version of the driver available on jcenter. Then import the RainbowHAT driver in the code.

```
// import the RainbowHat driver
import com.google.android.things.contrib.driver.rainbowhat.RainbowHat;
```

Let we see a sample usage of user driver for GPIO, PWM and I2C.

Light up a LED The following code can be used to access a led (the green one in this case) in Rainbow HAT board and turn on the light.

```
//open GREEN led connection.
Gpio led = RainbowHat.openLedGreen();
//turn on the light
led.setValue(true);
//close the device when done.
led.close();
```

The principal method of this "led user driver" is `openLedGreen()`. This method allows to open a connection to the GPIO2_IO00, the unique identifier that represents the green led embedded in the Rainbow HAT. The other two similar methods, for blue led and red led are `openLedBlue()` and `openLedRed()`.

We can notice that using these methods, a programmer does not have to remember the unique identifier for red led (GPIO2_IO02), blue led (GPIO2_IO05) and green led (GPIO2_IO00).

Detect when a button is pressed Also for the three capacitive buttons in the Rainbow HAT there are useful methods for access them.

The example shows how to open a connection to button "C" and detect when is pressed:

```
// Open a connection to button C
Button buttonC = RainbowHat.openButtonC();
// Set a listener for button events
buttonC.setOnButtonEventListener(new Button.OnButtonEventListener() {
    @Override
    public void onButtonEvent(Button button, boolean pressed) {
        Log.d(TAG, "button C pressed:" + pressed);
    }
});
// Close the device when done.
buttonC.close();
```

Through `openButtonA()`, `openButtonB()` and `openButtonC()` methods, programmers can have access to buttons "A", "B" and "C" (No knowledge of GPIO identifiers is required).

Play a note on the buzzer The usage of PWM buzzer is very easy to manage through the user driver. See the following code:

```
//open Piezo Buzzer connection.
Speaker buzzer = RainbowHat.openPiezo();
//Play the specified frequency. Play continues until stop() is called.
buzzer.play(600);
// Stop the buzzer.
buzzer.stop();
// Close the device when done.
buzzer.close();
```

The `openPiezo()` method return a `Speaker` object that represents the Piezo buzzer embedded in Rainbow HAT, and opens a connection to it. With the method `play()`, that takes a frequency as a parameter, it is possible to play a note on the buzzer at the chosen frequency.

Report the current temperature The `openSensor()` method allows to open a connection to the BMP280 sensor located on Rainbow HAT. Instead, `readTemperature()` method reports the current temperature registered by it.

```
// Open a connection to the BMP280 sensor
Bmx280 sensor = RainbowHat.openSensor();
sensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
// Report the current temperature
Log.d(TAG, "temperature:" + sensor.readTemperature());
// Close the device when done.
sensor.close();
```

Write on the display The following code shows the usage of the HT16K33 segment display to report the measured pressure.

```
// Open a connection to the BMP280 sensor
Bmx280 sensor = RainbowHat.openSensor();
sensor.setPressureOversampling(Bmx280.OVERSAMPLING_1X);
// Open a connection to the Ht16k33 display
AlphanumericDisplay segmentDisplay = RainbowHat.openDisplay();
// Set the brightness
segmentDisplay.setBrightness(Ht16k33.HT16K33_BRIGHTNESS_MAX);

// Set the text to show
segmentDisplay.display(sensor.readPressure());
// Enable text display
segmentDisplay.setEnabled(true);
// Close the devices when done.
sensor.close();
segmentDisplay.close();
```

In this case we have used `readPressure()` method, which reports the current pressure registered by BMP280 sensor. Through `openDisplay()` method, include in Rainbow HAT user driver, a connection to the `AlphanumericDisplay` Ht16k33 is opened. With the method `display()`, the text to show is set (string or number) and then with the method `setEnable(boolean b)` it is possible to enable the text display (boolean `true`) or disable it (boolean `false`).

In the previous code, we have seen the usage of Rainbow HAT user-drivers, which are those static methods called on the `RainbowHat` class (e.g. `RainbowHat.openSensor()` or `RainbowHat.openButtonC()`). However, methods belonging to specific components have been used too, for example `play()` method on the buzzer, `readTemperature()` and `readPressure()` on the BMP280 sensor, `display()` method on segment display.
All those methods, called over objects belonging to `AlphanumericDisplay` class, `Bmx280` class, `Speaker` class and `Button` class, are in the majority of cases included in the drivers avaiable in the *contrib-drivers* library on GitHub. These drivers greatly simplify the access and the management of the various components.

Chapter 3

Android Things and Architecture Components

In this chapter, we will present the **Android Architecture Components** and we will describe their usage in the Android Things app development, through a simple application for the temperature detection in the Rainbow HAT board. After that, we will discuss some relevant results obtained from the analysis of this application with the *Julia Static Analyzer*.

3.1 Android Architecture Components

The Android Architecture Components are a set of libraries announced by Google at *Google I/O 2017* as part of Android JetPack.[1] These libraries help programmers to develop more robust, maintainable and testable applications, moreover they define some important guidelines on how to structure the code during the development.

The reason that led to the publication of this library resides in the Android OS itself. Each Android application consists of different types of components that provide different services. There are the *Activity* and the *Fragment* necessary for the user interface, *Content Providers* which provide data sharing with other apps, *Services* that are useful for background tasks and also *BroadcastReceivers* needful for reacting to various events.
Each of these components follows its life cycle, which is strictly controlled by the OS. In fact, the application is subject to system decisions based on internal events and user behavior.
The applications do not have a monolithic structure nor does the execution always cross the same entry point, because some app components can be activated at different time, depending on the configuration received from the system.

[1] Android Jetpack is a suite of libraries, tools, and guidance to help developers write high-quality apps.

Another important reason concerns the application status: the application can be destroyed by the Android OS if needed to free up memory with the consequent loss of information.

The inspiring principle of the Android Architecture Components is **Separation of concerns**.
This pattern forces each component of the application to develop its expertises without creating bottlenecks. For example, programmers must avoid accumulating too many work within an Activity, but being able to discern which component to adopt, based on the operations to be performed.

The most important features introduced with the Android Architecture Components are:

- **Lifecycle-aware** components: they help programmers manage the Activity and Fragment lifecycles. They also survive configuration changes, avoid memory leaks and easily load data into UI.
- **LiveData** class: useful for creating observable objects that allow the exchange of data within the application.
- **ViewModel** class: the class on which the model is based, which will manage the data shown by the user interface.
- **Room**: a persistence library which provides an abstraction layer over SQLite in order to facilitate interactions with the database.
- **Paging**: a library that allows a gradual upload of data to satisfy the needs of the user, without penalizing the performance.

We will now focus on the description of `LiveData` and `ViewModel` class. These classes will be used in Section 3.2, where we will describe an Android Things application that takes advantage of the Android Architecture Components.

3.1.1 LiveData

LiveData is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.
LiveData considers an observer to be in an active state if its lifecycle is in the `STARTED` or `RESUMED` state. LiveData only notifies active observers about updates. Inactive observers registered to watch LiveData objects are not notified about changes [17].

Let we see how to create a custom LiveData class and its usage.

Extend LiveData The following sample code shows a `ButtonLiveData` class which extends the LiveData class. It includes three important methods:

- `onActive()`: it is called when the LiveData object has an active observer.
- `onInactive()`: it is called when the LiveData object does not have any active observers.
- `setValue()`: it updates the value of the LiveData instance and notifies any active observers about the change.

```
1  public class ButtonLiveData extends LiveData<Boolean>{
2
3    private static final String TAG = ButtonLiveData.class.getSimpleName();
4    private Button buttonA;
5
6     @Override
7     protected void onActive() {
8        super.onActive();
9        Log.d(TAG, "onActive");
10       try {
11            // Open a connection to button 'A'
12            buttonA = RainbowHat.openButtonA();
13            // Set an event listener for button pressure detection
14            buttonA.setOnButtonEventListener(new Button.OnButtonEventListener() {
15                      @Override
16                      public void onButtonEvent(Button button, boolean pressed) {
17                            // Notify the observer
18                            setValue(pressed);
19                      }
20             });
21        } catch (IOException e){
22                Log.e(TAG, "Unable to open Button A");
23        }
24    }
25    @Override
26    protected void onInactive() {
27       try {
28              // Close the connection to the button
29              buttonA.close();
30        } catch (IOException e){
31              Log.e(TAG, "OnInactive: "+e);
32        }
33         super.onInactive();
34         Log.d(TAG, "onInactive");
35    }
36 }
```

Listing 3.1: ButtonLiveData.java

This class implements a custom LiveData for button pressure detection in the RainbowHAT board. In `onActive()` method a connection with "button A" (`buttonA` variable) is opened; then using `setOnButtonEventListener()` method, it is set a listener to be called when a button event occurred. Into the `onButtonEvent()` method, the button pressure event is notified at each observer through the `setValue()` method.
In `onInactive()` method, the connection to the button "A" is closed with `tempSensor.close()`.

Work with LiveData The main steps to follow, in order to work with LiveData are:

1. The creation of a LiveData instance to hold a certain type of data. Usually, a LiveData object is stored within a ViewModel class and is accessed via a getter method.

```
// LiveData instance
private ButtonLiveData mButtonLiveData;

// Getter method that return a ButtonLiveData object
public LiveData<Boolean> getButtonLiveData(){
    if(mButtonLiveData == null)
        mButtonLiveData = new ButtonLiveData();
            return mButtonLiveData;
}
```

   The previous example shows the declaration of a ButtonLiveData instance, contained in the `mButtonLiveData` variable, and the getter method `getButtonLiveData()` that initializes the variable and then returns it as a `LiveData<Boolean>` object.
   The Android documentation suggests to store LiveData objects, that update the UI, in ViewModel objects in order to avoid bloated activities and fragments which are responsible for displaying data but not holding data state.

2. The creation of an Observer object. This object has to control what happens when the data of LiveData object change; for this purpose it must defines the `onChanged()` method.

```
private final Observer<Boolean> buttonLiveDataObserver =
    new Observer<Boolean>() {
        @Override
        public void onChanged(@Nullable Boolean pressed) {
            Log.d(TAG, "Button A pressed");
        }
};
```

Usually, the creation of an Observer object is done within the UI controller (e.g. an activity)

3. Binding of the Observer object to the LiveData object. This is accomplished through the `observe()` method that takes a LifecycleOwner object and the observer. This attachment is usually done in a UI controller, such as an Activity or Fragment.

```
public class MainActivity extends AppCompatActivity {

    // Instance of a ViewModel subclass
    private MainActivityViewModel mainActivityViewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ...

        // Binding between a LiveData object and its observer
        mainActivityViewModel.getButtonLiveData()
            .observe(MainActivity.this, buttonLiveDataObserver);
    }

    ...

}
```

Let suppose that MainActivityViewModel is the class which contains the ButtonLiveData instance and its getter method which were defined in point 1.
The previous piece of code declares a `mainActivityViewModel` variable, instance of MainActivityViewModel class and then, into the `onCreate()` method, performs the binding between the ButtonLiveData object (obtained with the instruction `myViewModel.getButtonLiveData()`) and its observer `buttonLiveDataObserver` defined in point 2.

In most cases, the `onCreate()` method of an app component is the right place to start looking at a LiveData object. Indeed, this ensures that the system does not make redundant calls from the `onResume()` method of an activity. This also ensures that the Activity has data that it can display as soon as it becomes active.

3.1.2 ViewModel

The ViewModel is a class designed to store and manage UI-related data in a lifecycle conscious way.

In the Android framework the components dedicated to the UI management are Activity and Fragment, which play the role of controller. The framework may decide to destroy or re-create a UI controller in response to certain user actions or device events that are completely out of programmer control. For example, some passages from one phase to the other of the life cycle induced by configuration changes (e.g. device rotation) result in the destruction of the Activity, which needs the subsequent reconstruction.

Every time the Activity is destroyed, the transient data is lost and their uploads are repeated if performed within the Activity. To avoid penalizing data consistency and application performance, according to the classical approaches, the application status should be saved and restored manually using the `onSaveInstanceState()` and `onRestoreInstanceState()` methods. However, this approach is only suitable for small amounts of data that can be serialized then deserialized, not for potentially large amounts of data.

The ViewModel API is perfectly suited to address this issue. A ViewModel is always created in association with an Activity (or a Fragment) and it will be retained as long as the Activity is alive. When this is destroyed for a configuration change (e.g. rotation) and then recreated, the new instance is reconnected to the existing ViewModel. So, the purpose of the ViewModel is to acquire and keep the information that is necessary for an activity. Usually, ViewModel exposes this information via LiveData object.

It is important to emphasize that ViewModel is only responsible to manage the data for the UI. It should never access the view hierarchy or hold a reference back to the activity.

The following `MainActivityViewModel` class implements a subclass of ViewModel.

```
1  public class MainActivityViewModel extends ViewModel {
2
3      private Gpio ledG; // GPIO instance for a led
4      private ButtonLiveData mButtonLiveData; // LiveData instance
5
6      public MainActivityViewModel(){
7          try{
8              // Open connection to red led
9              ledR = RainbowHat.openLedRed();
10         } catch (IOException e) {
11             Log.e(TAG, "Unable to open leds connection");
12         }
13         // Turn on the light
14          setGreenLight(true);
15     }
```

```
19
20      // Getter method that return a ButtonLiveData object
21      public LiveData<Boolean> getButtonLiveData(){
22
23          if(mButtonLiveData == null)
24              mButtonLiveData = new ButtonLiveData();
25
26          return mButtonLiveData;
27      }
28
29
30      public void setGreenLight(boolean value) {
31          // Turn the light
32          ledG.setValue(value);
33      }
34
35
36      @Override
37      protected void onCleared() {
38
39          if (ledG != null) {
40
41              // Turn off the light
42              setGreenLight(flase)
43
44              try {
45                  // Close the connection to the Green led
46                  ledG.close();
47              } catch (IOException e) {
48                  Log.e(TAG, "Unable to close the Green led);
49              }
50          }
51      }
52  }
```

Listing 3.2: MainActivityViewModel.java

It defines a ButtonLiveData object named `mButtonLiveData` and its getter method `getButtonLiveData()`; as we have seen in Section 3.1.1. In addition, it declares an instance of GPIO, represented by `ledG` variable, in order to communicate with the green led on RainbowHat. The public method `setGreenLight(boolean value)` allows to turn on/off the led depending on the boolean value passed. The opening of the connection with the green led is performed by the constructor, when the ViewModel class instance is created.
In this implementation it is important to note the **onCleared()** method: it is called when the ViewModel is no longer used and it is destroyed. In this case, inside of it, is performed the closure of the connection with the led.

It is possible to use the MainActivityViewModel class within an activity, in order to obtain a ButtonLiveData object for button pressure detection and then bind it to a specific observer for managing its notifications.

```
1  public class MainActivity extends AppCompatActivity {
2
3      // Instance of a ViewModel subclass
4      private MainActivityViewModel mainActivityViewModel;
5
6      // ButtonLiveData observer
7      private final Observer<Boolean> buttonLiveDataObserver =
8      new Observer<Boolean>() {
9
10         @Override
11         public void onChanged(@Nullable Boolean pressed) {
12
13             // Turn on the light when pressed
14             mainActivityViewModel.setGreenLight(true)
15         }
16     };
17
18
19     @Override
20     protected void onCreate(Bundle savedInstanceState) {
21         super.onCreate(savedInstanceState);
22
23         // Instantiation of MainActivityViewModel object
24          mainActivityViewModel = ViewModelProviders.of(this)
25                                      .get(MainActivityViewModel.class);
26
27         // Binding between a LiveData object and its observer
28         mainActivityViewModel.getButtonLiveData()
29                 .observe(MainActivity.this, buttonLiveDataObserver);
30     }
31
32     @Override
33     protected void onDestroy() {
34         super.onDestroy();
35         Log.d(TAG, "onDestroy");
36     }
37 }
```

Listing 3.3: MainActivity.java

The implementation of the observer and the binding with the corrisponding LiveData object are the same as seen in paragraph 3.1.1.

The two important aspects to notice now are:

- **AppCompatActivity** extension: the `MainActivity` class extends `AppCompatActivity` which extends `LifecycleOwner` interface.
- **ViewModelProviders** class usage: the instantiation of MainActivityViewModel object is done through the class `ViewModelProviders` (line 21) and not using constructor.

The following diagram shows the life cycle of ViewModel component:

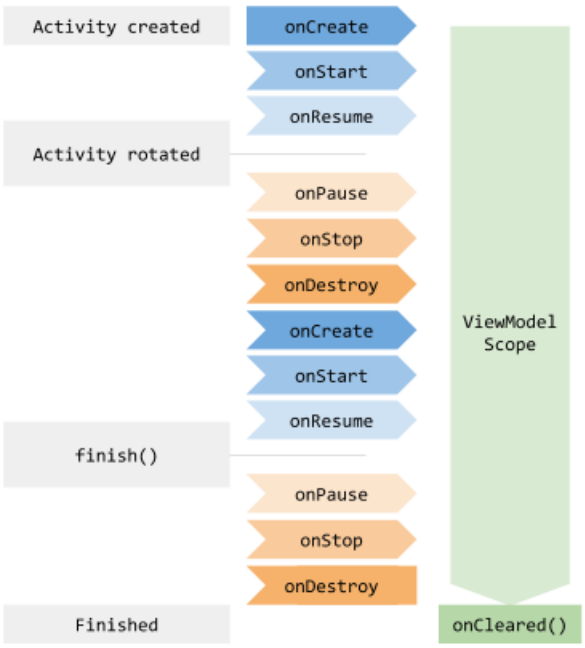


Figure 3.1: Lifecycles of Activity and ViewModel

A ViewModel object is bonded to the Lifecycle object passed to the ViewModelProvider, when getting the ViewModel. It remains in memory until the Lifecycle it's scoped to goes away permanently: therefore, in case of an Activity when it finishes.

3.2 Utilities in Android Things

In Android Things, part of the programming is focused on communication management with the available peripherals and on reacting to their changes. Every time a connection with a device is opened, the programmer has to worry about closing it when it is no longer necessary. This is important in order to free up resources, because of a new connection to the same device port cannot be open until the existing one is closed.
Furthermore, this opening/closing process must be sensitive to the Activity lifecycle.

It is now presented a simple Android Things application for the NXP Pico i.MX7D Starter Kit, which performs temperature detection.
The BMP280 sensor reports the current temperature every 2 seconds, which it is displayed using the alphanumeric display. The RainbowHAT LEDs are turned on according to the temperature value, in particular the application identifies three different temperature intervals:

- `temperature` $< 22°C$: blue led is turned on.
- $22°C \leq$ `temperature` $\leq 28°C$: green led is turned on.
- `temperature` $> 28°C$: red led is turned on and the alarm plays.

The detection is stopped when button "C" is pressed by the user and, before exiting the application, all the connection must be closed.

The flowchart 3.2 illustrates an high-level lifecycle of the application.

3.2.1 PicoTemperature_GodActivity

This first implementation of the application concentrates all the code and the operations in a single class: the `MainActivity`. This type of Activity, is usually called "*God Activity*".

In general, in object-oriented programming, a **God object** is an object that knows too much or does too much and is an example of an anti-pattern. A program that employs a God object has the most of its functionality coded into a single "all-knowing" object, which maintains most of the information about the entire program, and also provides most of the methods for manipulating this data. Because this object holds so much data and requires so many methods, its role in the program becomes God-like (all-knowing and all-encompassing) [18].

The entire code of PicoTemperature_GodActivity can be found in the appendix A. Now, we illustrate and comment the most important features.

First of all, the two constant `MAX_TEMPERATURE` and `NORMAL_TEMPERATURE`, for the different temperature threshold, are declared as global:

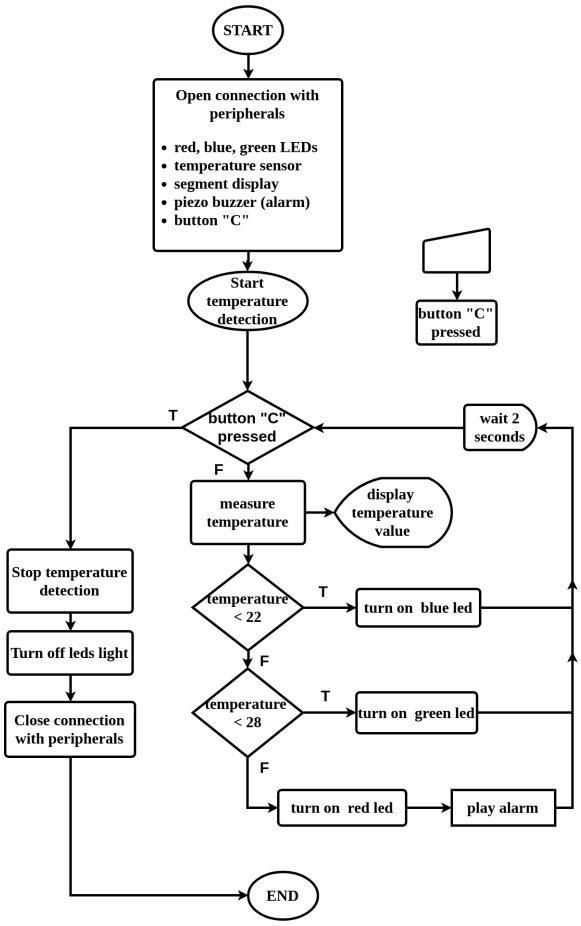


Figure 3.2: Temperature detection application flowchart

```
private static final float MAX_TEMPERATURE = 28.0f;
private static final float NORMAL_TEMPERATURE = 22.0f;
```

In addition, in the global scope are declared also the variables intended to manage the connection with the various peripherals; these variables are:

- `Bmx280 tempSensor`: the temperature sensor;
- `AlphanumericDisplay mDisplay`: the segment display temperature value observation;
- `Speaker buzzer`: the alarm;
- `Button buttonC`: the button for exit the application;
- `Gpio ledR, ledG, ledB`: the three LEDs which are turned on according to the temperature value;

In the `onCreate()` method, all the connections to the previous peripherals are opened and, since they are located in the RainbowHAT board, it is possible to use the *RaibowHAT device-drivers* as we have seen in Section 2.3.4.

The following pieces of code show only the opening phase of each connection.

Opening of LEDs connection:

```
ledR = RainbowHat.openLedRed();
ledG = RainbowHat.openLedGreen();
ledB = RainbowHat.openLedBlue();
```

Opening of temperature sensor BMP280 with `MODE_NORMAL` setting: this ensure the device is powered and not in sleeping mode before trying to read from it:

```
tempSensor = RainbowHat.openSensor();
tempSensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
tempSensor.setMode(Bmx280.MODE_NORMAL);
```

Opening connection to segment display, set brightness and enable the visualization:

```
mDisplay = RainbowHat.openDisplay();
mDisplay.setBrightness(HT16K33_BRIGHTNESS_MAX);
mDisplay.setEnabled(true);
```

Opening buzzer which will be our "alarm":

```
buzzer = RainbowHat.openPiezo();
```

In the `onCreate()` method it is also initialized an Handler object, called `myHandler`, which manages the temperature detection phase.

```
myHandler = new Handler(Looper.getMainLooper());
```

In order to use the button "C", it has been implemented a private method called `initButton()` that opens the communication with the button and registers a OnButtonEventListener object to capture when pressed.

```
buttonC = RainbowHat.openButtonC();
buttonC.setOnButtonEventListener(new Button.OnButtonEventListener() {
    @Override
    public void onButtonEvent(Button button, boolean pressed) {
        Log.d(TAG, "button C pressed");
        MainActivity.this.finish();
    }
});
```

When button "C" is pressed, the MainActivity is closed through the `finish()` method.

In `onStart()` method, it is started the temperature detection managed by the Handler:

```
protected void onStart() {
        super.onStart();
        // Start temperature detection
        myHandler.post(reportTemperature);
    }
```

The `reportTemperature` object is a Runnable object. In its `run()` mehod it registers the current value of temperature, it controls the LEDs light, it plays the alarm if necessary and it displays the temperature on the segment display. Before being displayed, the temperature value is appropriately scaled to two decimal places using `setScale()` method.

```
temperature = tempSensor.readTemperature();
BigDecimal tempBG = new BigDecimal(temperature);
tempBG = tempBG.setScale(2, BigDecimal.ROUND_HALF_UP);
temperature = (tempBG.floatValue());
```

The LEDs switching on/off is done through the method `setLedLight(char led, boolean value)` . It takes as parameters a `char` value (R,G,B allowed) and a `boolean` value; then it turns on or off the selected light (R = red, B = blue , G = green) depending on boolean value TRUE or FALSE.

Once the temperature value has been displayed and the LEDs have been managed, the Runnable object `reportTemperature` is stopped for two seconds before start again, using the instruction:

```
myHandler.postDelayed(reportTemperature, TimeUnit.SECONDS.toMillis(2));
```

Before exiting, all the connections previously opened must be closed, in order to free up the resources and make peripherals available to other applications.
This phase is managed within the `onStop()` method. All the connections are shut down by calling the `close()` method on the variables representing the peripherals.

3.2.2 PicoTemperature_AAC

This second implementation of the application introduces the Android Architecture Components (AAC) into the code.

In order to use the Architecture Components in an Android Things project, it is necessary to add the following Gradle dependencies in the application-level Gradle file:

```
dependencies {
      // Support to AppCompatActivity
    implementation 'com.android.support:appcompat-v7:27.1.1'
    // Support to LiveData and ViewModel
    implementation 'android.arch.lifecycle:extensions:1.1.1'
}
```

In addition, it is required to include in the `AndroidManifest.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  ... >

  <application
    android:theme="@style/Theme.AppCompat">
    ...
  </application>
</manifest>
```

First of all, it is possible to use the LiveData class in order to hide the management of the Button object from the MainActivity class. This transformation is very similar to the code example 3.1: also this class is named `ButtonLiveData` but now it is opened a connection to the button "C" (instead of "A") using the device-driver method `RainbowHat.openButtonC()`. Another small difference concerns the value that will be sent to the observers when the button value will change: every time the button "C" is pressed, a boolean value `TRUE` is delivered to each of them. A simple log is shown in the console for debug purpose.

```
buttonA = RainbowHat.openButtonC();
buttonA.setOnButtonEventListener(new Button.OnButtonEventListener() {
@Override
      public void onButtonEvent(Button button, boolean pressed) {
          Log.d(TAG, "button C pressed");
          setValue(true);
      }
});
```

This approach can be extended also to the temperature sensor BMP280: we want to hide from the MainActivity the creation/destruction cycle of its connection and the temperature detection management too. For this purpose we have implemented a LiveData class named `TemperatureLiveData`.

```
1  public class TemperatureLiveData extends LiveData<Float>{
2
3      private static final String TAG = TemperatureLiveData.class.getSimpleName();
4      private Bmx280 tempSensor;
5      private Handler myHandler;
6
7      private final Runnable reportTemperature = new Runnable() {
8
9        float temperature;
10
11       @Override
12       public void run() {
13
14         try {
15                 // Read temperature value from Bmx280 sensor
16                 temperature = tempSensor.readTemperature();
17
18                 // Convert and scale the value
19                 BigDecimal tempBG = new BigDecimal(temperature);
20                 tempBG = tempBG.setScale(2, BigDecimal.ROUND_HALF_UP);
21                 temperature = (tempBG.floatValue());
22
23                 // Notify the observer
24                 setValue(temperature);
25         }
26
27         catch (IOException | IllegalStateException e){
28                 Log.e(TAG, "Unable to read temperature");
29                 temperature = Float.valueOf(null);
30         }
31
32         myHandler.postDelayed(reportTemperature, TimeUnit.SECONDS.toMillis(2));
33       }
34     };
```

```
    @Override
    protected void onActive() {
        super.onActive();
        Log.d(TAG, "onActive");

        // Open a connection to the sensor
        try {
            tempSensor = RainbowHat.openSensor();
            tempSensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
            tempSensor.setMode(Bmx280.MODE_NORMAL);
        }
        catch (IOException e){
             Log.e(TAG, "Unable to open temperature sensor");
        }

        // Init the handler
        myHandler = new Handler(Looper.getMainLooper());

        // Start temperature detection
        myHandler.post(reportTemperature);
    }

    @Override
    protected void onInactive() {
        // Stop temperature detection
        myHandler.removeCallbacks(reportTemperature);

        // Close the connection to the sensor
        try {
            tempSensor.close();
        }
        catch (IOException e) {
            Log.d(TAG, "onInactive: " + e);
        }

        super.onInactive();
        Log.d(TAG, "onInactive");
    }
}
```

Listing 3.4: TemperatureLiveData.java

In the `onActive()` method, the connection to the sensor `tempSensor` is opened; then it is initialized an Handler object (variable `myHandler`) and it is set for the execution on the main thread. In the end, through the method `post(reportTemperature)`, the temperature detection is started. The Runnable object `reportTemperature` implements the temperature detection: the temperature is read using the `readTemperature()` method, then it is suitably converted and scaled and in the end, it is notified at each observer with the `setValue()` method. This procedure is executed every 2 seconds through the `postDelayed()` method, called on the Handler object, with the parameter `TimeUnit.SECONDS.toMillis(2)`.

In `onInactive()` method, the temperature detection is interrupted with the instruction `handler.removeCallbacks(reportTemperature)` and then the connection to the temperature sensor is closed (`tempSensor.close()`).
In this case, different from the previous approach, the `reportTemperature` Runnable object is only responsible for reading and communicating temperature value at each observer of the TemperatureLiveData class .

The choice of introducing the LiveData class, in order to control the button and the temperature sensor, it is justified by the fact that both components play an active role in the application lifecycle: they detect data changes (temperature or button pressure) and have to notify the application. Instead, other peripherals like the LEDs, the display and the buzzer play a passive role in the application, therefore the use of LiveData class to control them would not have produced great benefits.

Once introduced the previous classes, it was created a ViewModel class called `MainActivityViewModel`. Inside of it, we have implemented the two getter methods for the `ButtonLiveData` and `TemperatureLiveData` objects, which will be used by MainActivity class in order to obtain the related elements.

```
public LiveData<Boolean> getButtonLiveData() {
        return mButtonLiveData;
}

public LiveData<Float> getTemperatureLiveData(){
        return mTemperatureLiveData;
}
```

In this class we have also implemented other three public methods to control the various peripherals involved in the application lifecycle.
There is the `display(Float value)` methods which shows the passed value in the segment display; the `setLedLight(char led, boolean value)` method, for light on/off the different LEDs and the `playSound()` method which performs the buzzer ring.
Along with these, there are other private methods which open and close the peripherals connections. There are: `openLeds()` and `closeLeds()` methods for the LEDs, `openSpeaker()` and `closeSpeaker()` for the buzzer and in the end there are `openDisplay()` and `closeDisplay()` methods for the segment display.

The "open" methods are invoked in the constructor of MainActivityViewModel class, so the various connections are opened when the instance of the class is created. In the constructor are also created the two LiveData objects.

```
public MainActivityViewModel(){

        // Create LiveData
        this.mButtonLiveData = new ButtonLiveData();
        this.mTemperatureLiveData = new TemperatureLiveData();
        // Open a connection to the leds
        openLeds();
        // Turn off leds light when init the ViewModel
        setLedLight('R',false);
        setLedLight('G',false);
        setLedLight('B',false);
        // Open a connection to the display
        openDisplay();
        // Open a connection to the speaker
        openSpeaker();
}
```

Conversely, the "close" methods are invoked into the `onCleared()`, which is executed when the MainActivityViewModel is no longer used and it will be destroyed.

```
@Override
protected void onCleared() {
        // Turn off LEDs and stop the connection
        closeLeds();
        // Close the connection with the alarm speaker
        closeSpeaker();
        // Clear the display
        closeDisplay();
}
```

In the end, there is the MainActivity class, which is the entry point of the application. In this class are implemented the two observers for the LiveData objects: `exitButtonLiveDataObserver` and `temperatureLiveDataObserver`.

The `exitButtonLiveDataObserver` detects when the ButtonLiveData object observed sends to it a boolean value. If the value passed is `TRUE` the MainActivity is terminated, through the call of `finish()` method.

```
private final Observer<Boolean> exitButtonLiveDataObserver =
  new Observer<Boolean>(){
        @Override
        public void onChanged(@Nullable Boolean pressed){
            if (pressed){
                MainActivity.this.finish();
            }
        } };
```

The `temperatureLiveDataObserver` has to captures each value of temperature sent by the TemperatureLiveData object. After that, the tempera-

ture is shown in the segment display, through the `display()` method of the MainActivityViewModel class. Depending on its value, the LEDs are turned on/off, according to the different threshold reached and, just in case, the alarm is played.

```
private final Observer<Float> temperatureLiveDataObserver =
        new Observer<Float>() {

            @Override
            public void onChanged(@Nullable Float temperature) {
                mainActivityViewModel.display(temperature);

                if( temperature < NORMAL_TEMPERATURE){
                  mainActivityViewModel.setLedLight(R,false);
                  mainActivityViewModel.setLedLight(G,false);
                  mainActivityViewModel.setLedLight(B,true);
                } else if( temperature >= NORMAL_TEMPERATURE &&
                      temperature < MAX_TEMPERATURE){

                  mainActivityViewModel.setLedLight(R,false);
                  mainActivityViewModel.setLedLight(G,true);
                  mainActivityViewModel.setLedLight(B,false);
                } else {
                  mainActivityViewModel.setLedLight(R,true);
                  mainActivityViewModel.setLedLight(G,false);
                  mainActivityViewModel.setLedLight(B,false);

                  mainActivityViewModel.playSound();
                }
            }
};
```

In the `onCreate()` method it is initialized the MainActivityViewModel class instance, using the ViewModelProviders class; then through the `observe()` method, the observation of the LiveData objects is started.

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate");

    // Get MainActivityViewModel instance
    mainActivityViewModel = ViewModelProviders.of(this)
                                .get(MainActivityViewModel.class);
    // Start observing ButtonLiveData
    mainActivityViewModel.getButtonLiveData()
        .observe(MainActivity.this, exitButtonLiveDataObserver);
    // Start observing TemperatureLiveData
    mainActivityViewModel.getTemperatureLiveData()
        .observe(MainActivity.this, temperatureLiveDataObserver);
 }
```

The sequence diagram 3.4 demonstrates how the Android Architecture Components interact with each other and how they are sensible and respect the application lifecycle.
In particular, it shows the MainActivity creation and distruption and how the connection to the IoT peripherals are demanded to the different classes:

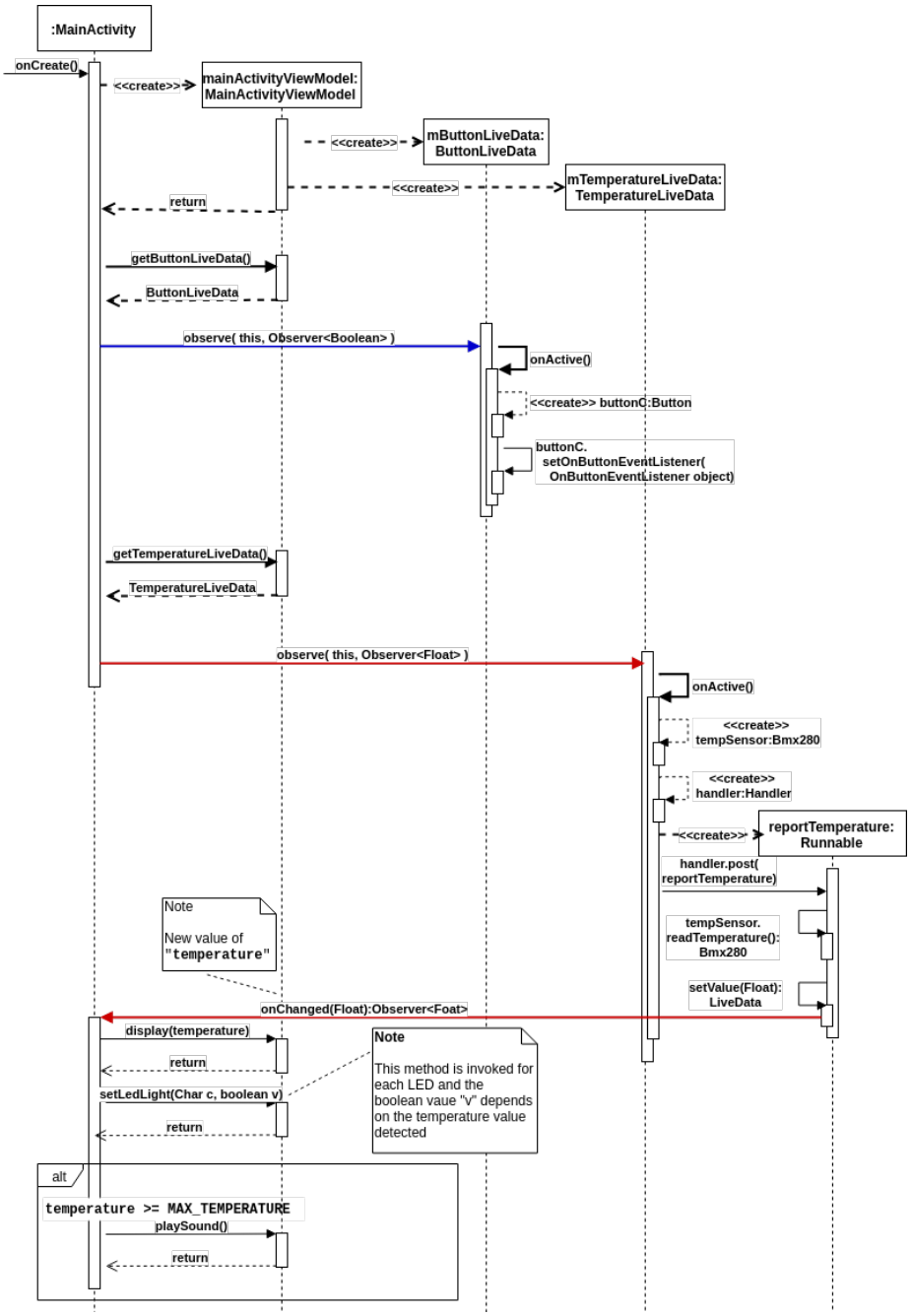
- after the MainActivity is created (`onCreate()`), a MainActivityViewModel instance is built and the LEDs, the display and the speaker conections are opened. Then, both LiveData components are activated (`onActive()`) and therefore they are ready to notify any changes;

- before the MainActivity is destroyed (`onDestroy()`), the LiveData objects are deactivated (`onInactive()`) and therefore no longer notify any changes. Then, the connection with the other peripherals are closed and in the end the Activity is destroyed (`onDestroy()`).

The figure 3.3 displays the output console of the PicoTemperature_AAC application. Besides being able to see the lifecycle of the application, it is possible to see how the LiveData objects and their observers are reactive to the change in the value detected.

```
D/MainActivity: onCreate
D/MainActivityViewModel: MainActivityViewModel instance created
D/MainActivityViewModel: Open LEDs connection
D/MainActivityViewModel: Open display connection
D/MainActivityViewModel: Open speaker connection
D/ButtonLiveData: onActive
D/TemperatureLiveData: onActive
D/MainActivity: onChanged() in temperatureLiveDataObserver: temperature = 23.13
D/MainActivity: onChanged() in temperatureLiveDataObserver: temperature = 23.19
D/ButtonLiveData: button C pressed
D/MainActivity: onChanged() in exitButtonLiveDataObserver: value = true
D/TemperatureLiveData: onInactive
D/ButtonLiveData: onInactive
D/MainActivityViewModel: Close LEDs connection
D/MainActivityViewModel: Close speaker connection
D/MainActivityViewModel: Close display connection
D/MainActivity: onDestroy
Application terminated.
```

Figure 3.3: Screenshot of the log console of PicoTemperature_AAC application


:MainActivity
onCreate()
<<create>>
mainActivityViewModel:
MainActivityViewModel
<<create>>
mButtonLiveData:
ButtonLiveData
<<create>>
mTemperatureLiveData:
TemperatureLiveData
return
getButtonLiveData()
ButtonLiveData
observe(this, Observer<Boolean>)
onActive()
<<create>> buttonC:Button
buttonC.
setOnButtonEventListener(
OnButtonEventListener object)
getTemperatureLiveData()
TemperatureLiveData
observe(this, Observer<Float>)
onActive()
<<create>>
tempSensor:Bmx280
<<create>>
handler:Handler
<<create>>
reportTemperature:
Runnable
handler.post(
reportTemperature)
tempSensor.
readTemperature():
Bmx280
setValue(Float):
LiveData
Note
New value of
"temperature"
onChanged(Float):Observer<Foat>
display(temperature)
return
setLedLight(Char c, boolean v)
return
Note
This method is invoked for
each LED and the
boolean vaue "v" depends
on the temperature value
detected
alt
temperature >= MAX_TEMPERATURE
playSound()
return

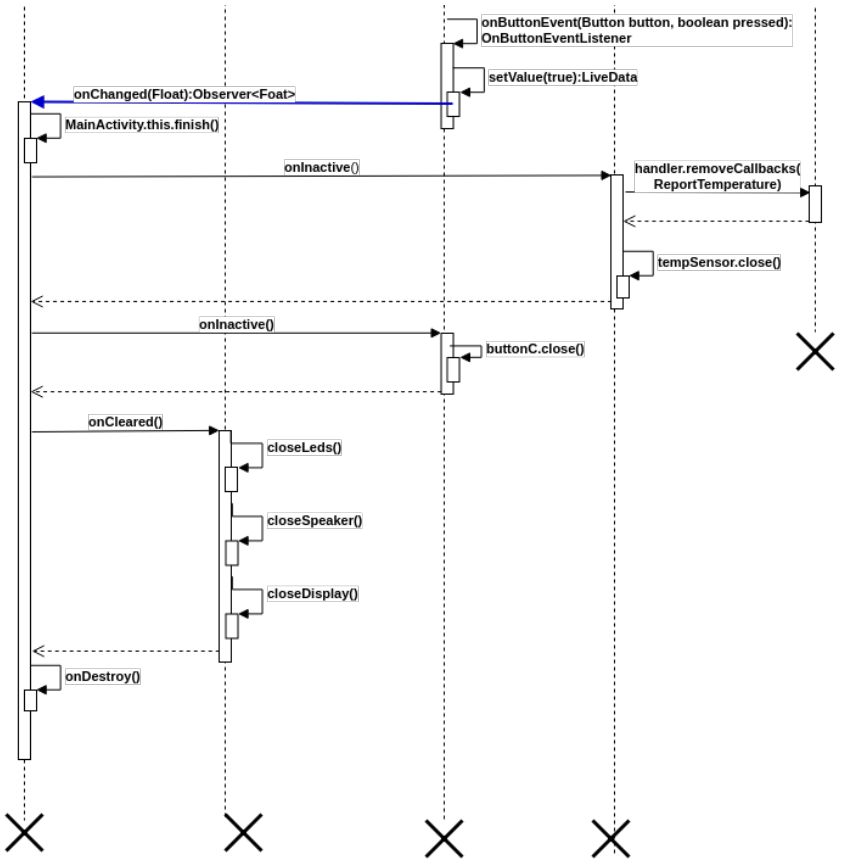


Figure 3.4: Sequence diagram for PicoTemperatureAAC

3.3 Analysis in Julia

After introducing the Android Architecture Components in the Android Things application code and after verifying how they adapt correctly to the programming requirements of Android Things, we have decided to operate some code analysis, using the Julia Static Analyzer.

The aims of these analyses are on the one hand to verify is if the developed application contains some critical bugs or error, on the other hand, is to test how the Julia Static Analyzer behaves when in the Android code are included the Android Architecture Components.

3.3.1 The Julia Static Analyzer

Julia is powerful tool of semantic static analysis for Java, Android and .NET, which permits to identify errors, bugs, security vulnerabilities and inefficiencies in the code. It is based on the abstract interpretation theory that guarantees a great accurancy and reliability of its results.

In geneal, in the software development lifecycle, there are two different approches that can be adopt in order to detectbugs and vulnerabilities:

- **Dynamic analysis**: experimental approach based on running the software.
- **Static analysis**: analytic approach based on the semantic model of the software.

In particular, static analysis is a type of software analysis that is performed without executing the program, unlike dynamic analysis.

A *Static Code Analyzer (SCA)* can be *syntactic* or *semantic*.
Syntactic SCAs check if some given patterns are locally found in the program, but they cannot prove the absence of errors because they do not perform any semantic correctness check of the code.
Semantic SCAs are based on formal methods such as mathematical theories to express and approximate the semantic of the programs.
Julia is the only commercial semantic SCA of Java and .NET code.

The Julia Static Analyzer can be used for different purposes:

1. **Find errors and inefficienties** in the code, in order to fix them at the initial phase of software development lifecycle. This ensures the improvement of the quality of the application.
2. **Analyze third-part code**, in order to evaluate the quality of applications whose source code is not avaiable. This is done using only the bytecode of the application.

Julia is the only tool for Java, Android e .NET to implement analysis for properties like *nullness* and security such as *SQL-injection, XSS* and *privacy leaks* like GDPR. Other tools on the market analyze the code, issue warnings of some vulnerabilities but do not report many others, because the do not have the *algorithmic depth* to detect them, since they are based mostly on syntactic pattern-matching techniques.

Julia includes a complete range of checkers for quality, efficiency, security and style issues. The tool divides the various checkers in two main categories: **basic** for a simple identification of bugs or errors and **advanced**, for a more complex and accurate analysis.

3.3.2 Analyses

We have executed the analysis using two different checkers: first the basic **Close Resource Checker**, then the advanced **GDPR Checker**. We now report the achieved results.

Close Resource Checker

This checker belongs to the **Basic** checkers group and identifies leakage of resources left open.

In general, resources such as files or database connections should be closed, or otherwise the system might run into an out of resource exception. The close operation should always be performed, in every execution path. For this reason, the `try-with-resource` or `finally` constructs should be used. This checker verifies that resources are always closed after being open, for every execution path. This checker recognizes and accepts as correct code where a resource is stored into a field or returned from a method and is then closed at the end, at least for some frequent scenarios [19].

After performing the analysis, Julia identifies **7 minor bugs**.

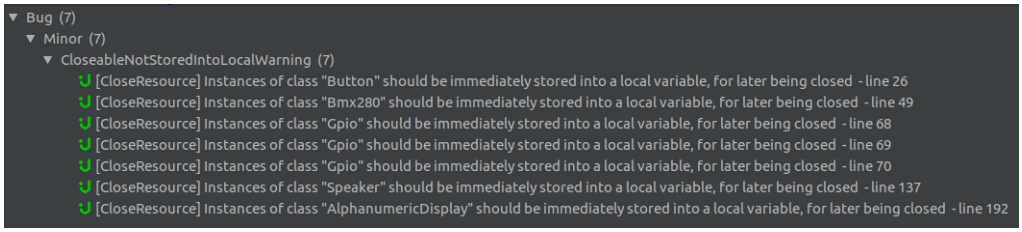


Figure 3.5: Julia analysis results in AndroidStudio

The warnings belong to `CloseableNotStoredIntoLocalWarning` and Julia reports the following related information:

`warningDescription:`	`a closeable has not been immediately stored into a local variable.`

This warning is related to all those Closable objects that are opened and closed in two different methods.
Indeed, in the MainActivityViewModel constructor, there are the Gpio, the Speaker and the AlphanumericDisplay objects whose connections are opened through some dedicated methods, for example `openLeds()` rather than `openSpeaker()`. Then, when the `onCleared()` method is invoked (before destroying the MainActivityViewModel instance) all the "close()" methods of each Closable object are called.
Also into the LiveData classes the connection to the peripherals (`Button` and `Bmx280` objects) is opened in `onActive()` method and the related closure is performed into the `onInactive()` method.

To avoid these warnings, a possible solution is to open and close each connection with the peripherals, every time they are used; thus in the MainactivityViewModel, perform the connection management in the public methods `setLedLight()`, `playSound()` and `display()`.
Instead, in the LiveData classes, the connections with the temperature sensor and even more with the button "C", must necessarily remain open: remember that the application observes the button "C" and detects when it is pressed, therefore it is essential to maintain an active connection with it.

To observe more closely the behavior of the CloseResource checker, when the Android Architecture Components are used, we have developed a simple testing application, which partly reproduces the behavior of the PicoTemperature_AAC.
As mentioned above, into the LiveData classes the opening of the connections to the peripherals (Closeable objects) takes place in the `onActive()` method, while their closure occurs in the `onInactive()` method.
Therefore, with the testing application, we want to study the behavior of the checker when the opening/closing cycle with these Closeable objects occurs in two separate methods.

PicoPiComponentTest

The testing application is called PicoPiComponentTest. The application consists of a single Activity and two methods: `onCreate()` and `onDestroy()`. The application focuses on the usage (opening and closing) of the `mI2CDevice` variable, which is an I2CDevice object.

`mI2CDevice` is declared global:

```
private I2cDevice mI2cDevice;
```

In the `onCreate()` method, the connection is opened:

```
try{
    PeripheralManager mPeripheralManager = PeripheralManager.getInstance();
    mI2cDevice = mPeripheralManager
    .openI2cDevice(DEFAULT_I2C_BUS, DEFAULT_I2C_ADDRESS);
}
catch(IOException e){...}
```

In the `onDestroy()` method, the connection is closed:

```
try {
    mI2cDevice.close();
} catch (IOException e) {...}
```

The Julia Analyzer gives, as expected, a `CloseableNotStoredIntoLocalWarning`:

`warningDescription:`	`a closeable has not been immediately stored into a local variable.`
`warningMessage:`	`instances of class "I2cDevice" should be immediately stored into a local variable, for later being closed.`

Alternatively, if the variable `mI2CDevice` is created locally to the `onCreate()`, it becames no longer visible to the `onDestroy()` method and it is not possible to perform its closure within it.

When an output device is used, the general strategy is to open the connection to the specific component (let suppose of a generic type `MyClosable`) in the `onCreate()` method and then assign its instance to a variable, for example `myClosable`. Then, set the value for the component, calling its appropriate method (e.g. `myClosable.setValue()` and in the end close the connection.

Before exiting the application, in the `onDestroy()` method, it is necessary to reopen the connection with the component, by assigning the new instance to the a variable `myClosable_bis`, set the desired exiting value for the component and then close the connection again.

With reference to the PicoTemperature_AAC application 3.2.2, this procedure is restricted to some output components, for example a led or an

alarm, which once set a value keep it until a new one is set again.
On the other side, for output components like temperature sensors or input devices like buttons, whose measured values need to be read continuously, extending this procedure would not be feasible.

GDPR Checker

This checker belongs to the **Advanced** checkers group and identifies privacy issues for GDPR compliance, and produces a detailed report.

The European GDPR (General Data Protection Regulation) directive will take the companies responsibilities to a new level introducing stricter rules and sanctions for non-compliancy. Protecting sensitive data is a top priority when it comes to information management. Hence Julia provides this checker in order to support GDPR compliance and to track explicit information flow in the program, from manually selected sensitive data locations into manually selected leakage locations. In this way, Julia allows one to have an exhaustive report to identify potential data leaks, with a soundness guarantee derived from the use of abstract interpretation, and with the freedom to bend the analysis to specific needs, through the use of code annotations [20].

The aim of this analysis is to trace the flow of some critical informations managed by the application. This time, the intention is verify how the GDPR checker works in an Android Application that uses the Android Architecture Components.

The GDPR checker is composed by two phases: **Init** and **Report**.

The *Init* phase allows to create a Specification File from the application code analyzed. It contains the possible candidate points that can become leakage locations and possible sensitive data sources locations. Into this file it is possible to tag the program points to analyze and eventually set up a policy with allowed data flows.
These are the tag names for our sensitive data and leakage points:

- Type of sensitive data: **temperature**
- Type of leakage point: **star**

Then, we have defined as sensitive data the value returned by the following method:

- **readTemperature()**: belongs to Bmx280 contrib-drivers class. It returns a Float value which represents the temperature detected by the RainbowHat sensor BMP280.

In the end, we have specified the following methods as *"sink"* points:

- **setValue(Object o)**: belongs to LiveData class. This method sends the object `'o'` to the LiveData class observers.
- **display(Float value)**: belongs to MainActivityViewModel class. This method print into the segment display the passed value.
- **display(Double n)**: belongs to AlphanumericDisplay class. This method print into the segment display the passed value.

In this case no policy is defined. Each flow of sensitive data towards the sink points is considered as not allowed.

The *Report* phase has revealed **2 bugs**, whose type is `LeakageOfPrivateData ThroughParameterUnknownSourceWarning`.

This warning reports the following description:

```
warningDescription:  some privata data from an unknown source is
                     leaked through a parameter of a method call
```

These are the critical flows detect:

1. Flow of temperature value, passed as a parameter to the `display()` method of the AlphanumericDisplay class.
2. Flow of temperature value, passed as a parameter to the `display()` method of the MainActivityViewModel class.

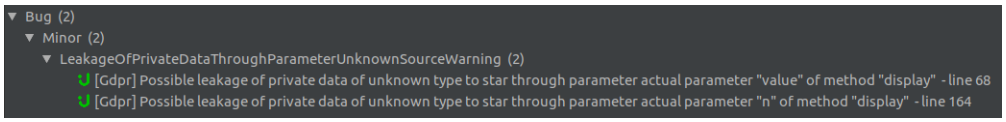


Figure 3.6: Julia GDPR-checker analysis results in AndroidStudio

With reference to the code 3.4, where the `setValue()` method of LiveData class is used, we can observe that Julia can not identify another critical flow evidently present in the code:

- Flow of the value returned by the `readTemperature()` method, called in the TemperatureLiveData class (line 15) and then passed as a parameter to the `setValue()` method (line 23).

This time we have pointed out an unusual behavior of Julia Static Analyzer when analyzing the information flow through the Android code which makes use of Android Architecture Components.

This behavior has been revealed also in the ButtonLiveData class, where the `setValue()` method is invoked when the button "C" is clicked and it sends a boolean value to all the observers.

This inability of the analyzer to identify the flow of information between a LiveData object and its observer, through the `setValue()` method, could be potentially harmful.

In our code, the `temperatureLiveDataObserver` observes a single LiveData object. After the temperature value reaches the observer, it is passed to the `display()` method, invoked just inside the observer and in this case Julia correctly identifies this stream as not allowed, triggering a warning: therefore, from the observer's side, the analyzer is able to perceive the flow of information between the LiveData and the observer.

However, the Android Architecture Components allow an observer to combine multiple LiveData objects into a single LiveData and receive updates whenever data sources are updated: this is done via *MediatorLiveData*[2].
In this scenario, one possible policy is to mark only certain LiveData objects as sensitive data sources. Therefore, being able to identify from which LiveData the sensitive flow arrives, that is to understand in which LiveData the method `setValue()` receives sensitive information, is crucial.

[2]LiveData subclass which may observe other LiveData objects and react on OnChanged events from them.

Chapter 4

Bluetooth Low Energy

In this chapter we analyze the use of Bluetooth Low Energy (BLE) API for Android Things. We have developed two applications which perform the interaction between the NXP Pico i.MX7D Starter Kit and an Android smartphone using the BLE protocol. The applications, presented in Section 4.2, have two opposite architectures and through their study we have detected a relevant bug in the BLE API.

4.1 Description

4.1.1 The BLE technology

Bluetooth Low Energy (Bluetooth LE, colloquially BLE, formerly marketed as Bluetooth Smart) is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group (Bluetooth SIG) aimed at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries [21].

This technology is designed to be power-efficient, focuses to be simple and less performance-oriented and it is intended to reduce the costs while maintaining a similar communication range as Bluetooth.
Besides having "Bluetooth" in the name, the BLE protocol does not share much more with previous Bluetooth versions. It still works on 2.4 GHz ISM band but it has a new RF stack, it uses a simpler modulation system and the chip are smaller, cheaper and less energy hungry. This key characteristic turned out to be the catalyst for the explosion of a wide assortment of new IoT devices and applications on the market, for example wearables, lightbulbs, sensors, socks, cups, medical devices, and other smart-products.

4.1.2 Communication between device

Bluetooth Low Energy communication between device and mobile application follows usually a scheme [5]:

1. Device (peripheral) broadcasts an advertisement.
2. Central device (mobile phone) scans for advertisements.
3. Once the specific advertisement packet is received, the central device stops scanning, and initiates a connection to the broadcasting peripheral.
4. Central device browses the peripheral device for available services.
5. Central device exchanges information with peripheral device using characteristic read/write/notify requests and responses.

Broadcast advertisement. The *peripheral* device makes the advertisement sending packets with a specified interval and TX power level. The advertisements are broadcasted using three dedicated channels, whose frequency is optimized to avoid interferences with WiFi signal. The advertised packets are very small (31 bytes) and they have a specific formatting defined by Bluetooth SIG[1]. The broadcast packet is by design visible to all listening devices in range.

Listening for the advertisements. The *central* device, which has activated the scanning advertisement mode, receives all the advertisements of nearby devices. Then, the mobile application matches the received advertisements with a specific one, related to given known device. The scanning advertisement mode requires a lot of power to be performed, thus the central device usually stop the scanning immediately after receiving the first matching packet.

Connection to a device. Depending on the usage scenario, the mobile application may handle only advertisements, without initiating a connection. However, if more data exchange is needed, a connection is opened and it is addressed to the MAC address of a device having a matching advertisement.

As we can notice, there is a distinction between *central* device and *peripheral* device. In general, in the BLE protocol there are different types of roles, depending on whether we consider the connection phase or the talk phase of the devices.

[1]The Bluetooth Special Interest Group (Bluetooth SIG) is the standards organisation that oversees the development of Bluetooth standards and the licensing of the Bluetooth technologies and trademarks to manufacturers.

- **central** and **peripheral**: regarding the BLE connection phase.
  - The device in the *central* role (*Master*) scans and looking for advertisement. It also can initiates an outgoing connection request to an advertising peripheral device.
  - The device in the *peripheral* role (*Slave*) makes the advertisement and waits for incoming connection request.
- **GATT server** and **GATT client**: regarding the devices talk phase. This determines how two devices talk to each other once they've established the connection, in order to exchange data.
  - The *Gatt client* is a device which accesses remote resources over a BLE link using the GATT protocol. Usually, the master is also the client.
  - The *Gatt server* is a device that has access control methods and provide resources to the remote client. Usually, the slave is also the server.

The GATT functionality of a device is logically separate from the master/slave role. The master/slave roles control how the BLE radio connection is managed, instead the client/server roles are dictated by the storage and flow of data [25].

Browsing the device services. After the connection has been established, the central device scans the peripheral in order to find all available services, characteristics and descriptors.

GATT data structure Devices exchange data using General Attribute Profile (GATT), characteristics, descriptors and services.

- **Generic Attribute Profile (GATT)**: the GATT profile is a general specification for sending and receiving short pieces of data known as "attributes" over a BLE link. All current Low Energy application profiles are based on GATT [22].
- **Attribute Protocol (ATT)**: it is a specification which defines the *Attribute Protocol*, that is used for discovering, reading, and writing attributes on a peer device.
  An attribute is uniquely identified by a UUID[2] and it is formatted as characteristics and services.
- **Characteristic**: a characteristic is composed by a single value and zero or more descriptors. The characteristic contains some properties

[2] 128-bit number used to identify information in computer systems

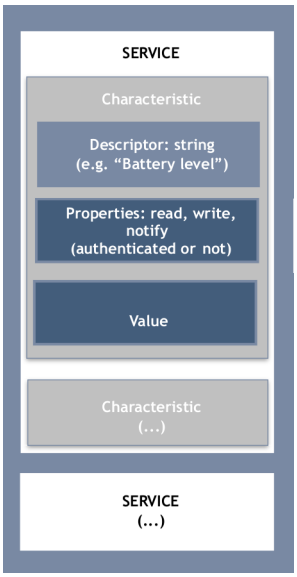


Figure 4.1: Representation of GATT data structure and relation between its components

that define its possible actions: read, write, notify. Each properties can be used separately or in a combined way.

- **Descriptor**: it gives additional information about a characteristic.
- **Service**: it is a collection of characteristics that work together in order to provide a specific functionality (e.g. heart rate measurement).

4.1.3 Security

The Bluetooth Low Energy specification provides several features to cover the encryption, trust, data integrity and privacy of the user's data. The most relevant aspects are:

- encryption;
- random MAC address;
- whitelisting.

Encryption and Pairing

Two BLE devices that want to communicate through an encripted transmission must undergo a pairing procedure.

The pairing mechanism is the process where the devices involved in communication exchange their identity information to set up trust and get the

encryption keys ready for the future data exchange. Bluetooth has several options for pairing depending on the capability of the device and the user's requirement[6].

The Figure 4.2 illustrate a BLE pairing. When the pairing starts, the two devices agree on a Temporary Key (TK), whose value depends on the pairing method used. After that, the devices exchange two random values and generate a Short Term Key (STK), that is calculated from the random values and the TK. The link is then encrypted using the STK.

This encription permits a secure key distribution; indeed, in BLE it is used a pairing key transport rather than a key agreement.
The key exchange involvs three keys: the *Long Term Key (LTK)* used for secure consecutive connections, the *Identity Resolving Key (IRK)* ensures device identity and privacy and the *Connection Signature Resolving Key (CSRK)* provides authentication of unencrypted data.

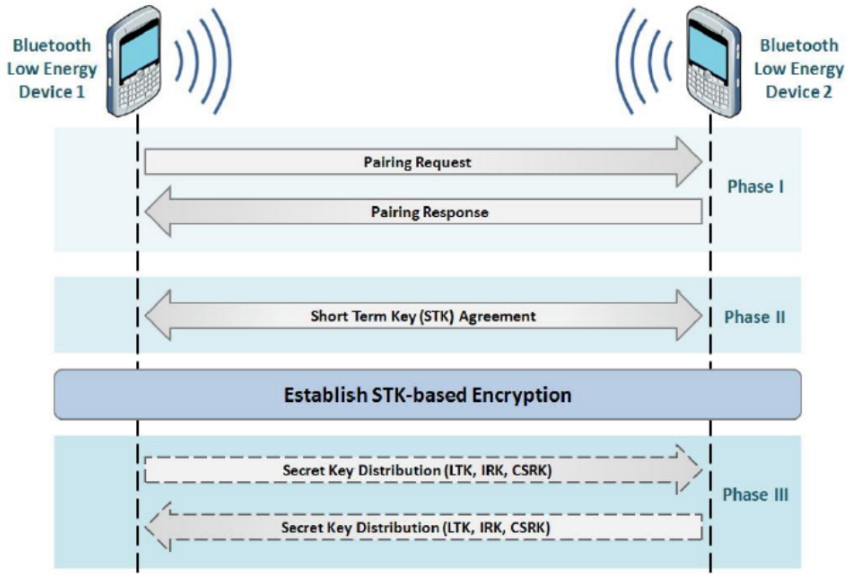


Figure 4.2: Bluetooth Low Energy Pairing

Random MAC address

The BLE specification allows the change of the device MAC address with a certain frequency. The aim of this procedure is to prevent tracking. Only two paired devices are able to resolve their related MAC addresses.

Whitelisting

In BLE it is possible to make a list of accepted devices MAC addresses. This *whitelist* will contain all the remote devices that are allowed to communicate

with the local device.

4.1.4 Bluetooth LE in Android Things

The Android APIs provides two packages for the Bluetooth LE usage: `android.bluetooth` and `android.bluetooth.le`.
The first contains classes, methods and constants for the use of both Bluetooth and BLE; the second refers only to the BLE, but is only available for API 21 or greater (Android 5.0).

For the Android Things operating system, a specific library is also available to manage Bluetooth functions, named `com.google.android.things.bluetooth`: this one provides functionalities for classic Bluetooth and for Bluetooth LE.

Usually, during the development of an Android Things application that uses Bluetooth features, it is necessary to use both the Android and the Android Things Bluetooth libraries. This mix of libraries and lack of good accompanying documentation makes non-trivial development of even a simple application.

We now describe the most important aspects that need to be kept in mind when developing an Android Things app that uses Bluetooth Low Energy. This aspects will be useful in the next section, where we will present two of these applications.

BLE permissions

In order to use Bluetooth in an application, it must be declared the `BLUETOOTH` and the `BLUETOOTH_ADMIN` permission in the Manifest file.

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

In addition, it is necessary to add the following permission (from the Android Things library), in order to use the Bluetooth connection and pairing APIs:

```
<uses-permission android:name="com.google.android.things.permission
                                        .MANAGE_BLUETOOTH" />
```

Set up BLE

The *BluetoothAdapter* class represents the local device Bluetooth adapter (the Bluetooth radio). It is required for any and all Bluetooth activity.

There is one Bluetooth adapter for the entire system, and any application can interact with it using its instance.

To get a BluetoothAdapter object it is necessary to create a BluetoothManager instance through the `getSystemService()` mehod and then use the `BluetoothManager.getAdapter()` function on BluetoothManager.

The `enable()` method allows to turn on the adapter, instead the `disable()` method permits to turn it off. To check whether Bluetooth is currently enabled there is the `isEnabled()` method.

Find BLE devices

The BLE devices search is executed using the `startLeScan()` method. This method takes a `BluetoothAdapter.LeScanCallback` as a parameter. The `stopLeScan()` method stops the scan. The callback must be implemented, because defines how scan results are returned.

This is an example of how to perform a scan:

```
private BluetoothAdapter bluetoothAdapter;
private boolean mScanning;
private Handler handler;
// Stops scanning after 10 seconds.
private static final long SCAN_PERIOD = 10000;

private void scanLeDevice(final boolean enable)
    if (enable) {
        // Stops scanning after a pre-defined scan period.
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                mScanning = false;
                bluetoothAdapter.stopLeScan(mLeScanCallback);
            }
        }, SCAN_PERIOD);
        mScanning = true;
        bluetoothAdapter.startLeScan(mLeScanCallback);
    } else {
        mScanning = false;
        bluetoothAdapter.stopLeScan(mLeScanCallback);
    }
}
```

Because scanning is battery-intensive, it is important to stop it as soon as the desired device is found or set a time limit on the scan, so never scan on a loop.

The following code implements a LeScanCallback. When a specific device is found (`ANDROID_DEVICE_SEARCHED`) the mRemoteBluetoothDevice object which represents it and its address are saved respectively in the variables `mRemoteBluetoothDevice` and `mRemoteDeviceAddress`.

```
private static final String ANDROID_DEVICE_NAME = "my_ble_device";

private BluetoothAdapter.LeScanCallback mLeScanCallback =
    new BluetoothAdapter.LeScanCallback() {

    @Override
     public void onLeScan(final BluetoothDevice device, int rssi, byte[] scanRecord) {
         Boolean isDeviceFound = false;
             if(device != null){
                 final String deviceName = device.getName();
                 if (deviceName != null && deviceName.length() > 0) {
                     Log.d(TAG, "Found device: "+deviceName);
                     if(deviceName.equals(ANDROID_DEVICE_SEARCHED)){
                         mRemoteBluetoothDevice = device;
                         mRemoteDeviceAddress = device.getAddress();
                     }
                 }
             }
    }
};
```

I/O capabilities

The BluetoothConfigManager class allows to report the input/output capabilities of the device. The `setLeIoCapability()` method, which provides this functionallity, accepts one of the following values:

- **IO_CAPABILITY_NONE**: Device has no input or output capabilities. This is the default value.
- **IO_CAPABILITY_OUT**: Device has a display only.
- **IO_CAPABILITY_IN**: Device can accept keyboard user input only.
- **IO_CAPABILITY_IO**: Device has a display and can accept basic (yes/no) input.
- **IO_CAPABILITY_KBDISP**: Device has a display and can accept keyboard user input.

As reported in the official documentation " *Currently Android Things devices reporting `IO_CAPABILITY_IN` or `IO_CAPABILITY_KBDISP` are not capable of pairing with a remote device that also reports `IO_CAPABILITY_KBDISP`, such as an Android mobile device* [23]".

BluetoothConnectionManager class

The *BluetoothConnectionManager* class belongs to the `com.google.android.things.bluetooth` library and it is responsible for handling Bluetooth pairing and connections with a remote Bluetooth device (BluetoothDevice object).

The main operations that can be managed with this class are[24]: initiate pairing with a new device, remove existing bond with a known device, control the pairing flow via user consent/input, and confirm pairing for outgoing and incoming pairing requests, establish profile connections with a remote Bluetooth device, disconnect existing profile connections with a remote device and obtain user consent for incoming connection requests and confirm/deny them.

Pairing with a remote device

The steps to follow in order to start the pairing process are:

1. Register a *BluetoothPairingCallback* [3] with the BluetoothConnectionManager.

2. Call the `initiatePairing()` method with the discovered peer device.

3. If the peer device reports the IO_CAPABILITY_NONE, the pairing process proceeds automatically. Once pairing is complete, a callback will be received in the `onPaired()` method of the BluetoothPairingCallback.

   Otherwise, if the peer device requires user input to pair, it is necessary handle the pairing request in the `onPairingInitiated()` method of the BluetoothPairingCallback.

Connecting to a remote device

Once the pairing procedure has been successfully completed, it is possible to connect with the profiles and services on the remote device. The connection to a specific profile on a given BluetoothDevice requires the following operations:

1. Register a *BluetoothConnectionCallback* [4] with the BluetoothConnectionManager.

2. Initiate the connection with the `connect()` method, providing the device and the target BluetoothProfile.

3. Handle the connection request in the `onConnectionRequested()`.

[3] Callback that is invoked during the Bluetooth pairing process. It contains all the relevant pairing information required for pairing.

[4] Callback that is invoked when the device receives requests for Bluetooth profile connections. It contains relevant information to handle the connection process.

4.2 Two use cases

In this section we present two implementations which use the Android Things BLE APIs. Through these apps and their testing, we have identified some bugs affecting the APIs that will be presented in Section 4.3.

4.2.1 The BLEPicoServer application

The first project we have developed is the *BLEPicoServer*.

It consists of two applications:

- **BLEserver**: the main Android Things app;
- **BLEclient**: a companion Android app.

The BLEserver app runs in the NXP i.MX7D Starter Kit (a.k.a PicoPi); the BLEclient app runs in an Android smartphone (a Samsung Galaxy S8 SM-G950F runs Android 9).

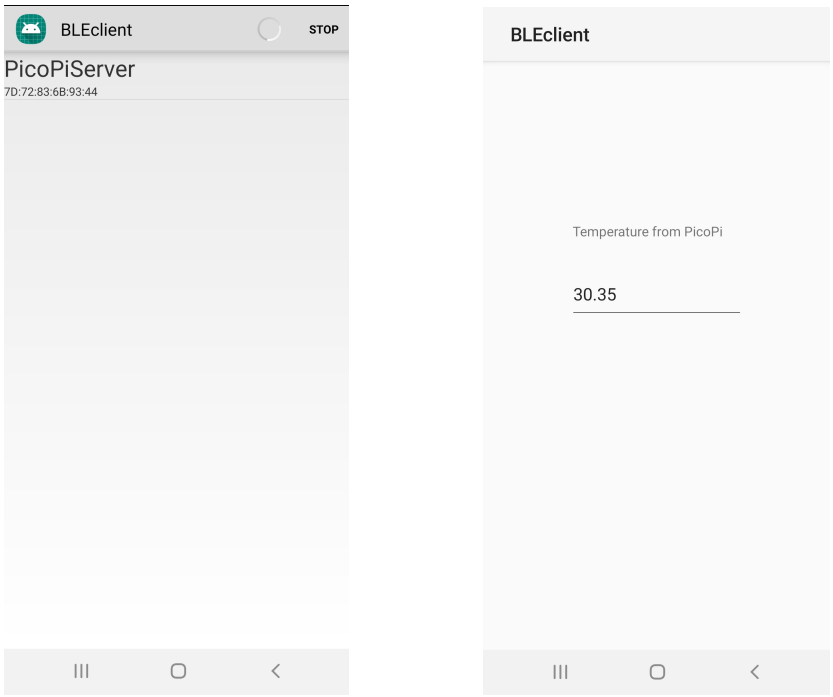
The role of *Master* is played by the smartphone: it starts scanning for available devices that advertise, in order to start a connection. The list of accessible devices is displayed in the smatphone and, when a device in the list is selected by the user, a connection is initialized: if the devices were already paired, the connection proceeds normally, else a pairing phase begins, with a pairing request sent by the smartphone.

The role of *Slave* is played by the PicoPi: it broadcasts information about its service and its characteristics and it waits for the incoming connection requests. After the connection is established, the PicoPi starts sending the temperature value, detected by its sensor, over the BLE channel.

BLEclient

The BLEclient is a simple Android app which includes two activity: *DeviceScanActivity* and *MainActivity*.
The first manages the scan of the available devices and reports them in a simple menu, the second manages the connection with a selected device (it manages also the pairing process if necessary) and then it displayes the temperature value received from it.
In addition, it contains the *BluetoothLeService* class which implements an Android Service for managing the connection and data communication with a GATT server hosted on a given Bluetooth LE device (PicoPi in our case). This class is provided by Google and it is available in the *googlesamples* repository on GitHub [5].

[5] https://github.com/googlesamples/android-BluetoothLeGatt



(a) DeviceScanActivity (b) MainActivity

Figure 4.3: BLEclient application

We do not dwell much on the implementation of this app, which is available in the appendix A; we point out only two aspects.
The first one concerns the scan function implemented in the DeviceScanActivity: it is very similar to the `scanLeDevice()` procedure shown in Section 4.1.4, but this time we have adopted the `startScan(ScanCallback s)` method, belongs to the BluetoothLeScanner class, instead of the `startLeScan(LeScanCallback s)` method from the BluetoothAdapter class. Indeed, from API level 18 and greater the latter is deprecated.

The second aspect we stress regards the pairing process managed by the MainActivity. In this case the application uses the Android APIs, so the pairing phase with the remote device is started through the `createBond()` method, belongs to the BluetoothDevice class, instead of the `initiatePairing()` method presented in Section 4.1.4 and belonging to the Android Things APIs.

BLEserver

The BLEserver app is composed of a single activity named *MainActivity* and it includes the *RemoteSensorProfile* class. The RemoteSensorProfile is the class which represents the collection of services offered by the

PicoPi. In this case there is only one service identified by the constant `REMOTE_SENSOR_SERVICE_UUID`. This service contains a single characteristic identified by the `TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID`.

```
1  public class RemoteSensorProfile {
2
3      private final static String TAG = RemoteSensorProfile.class.getSimpleName();
4
5      // Remote Sensor Service UUID
6      public static UUID REMOTE_SENSOR_SERVICE_UUID =
7      UUID.fromString ("B340B65C-B8AE-49E7-8ED8-F79C61708475");
8      // Remote Sensor Data Characteristic UUID
9      public static UUID TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID =
10     UUID.fromString("FEE891B9-032A-43AF-8923-5E3A4FF989A3");
11
12     public static BluetoothGattService createRemoteSensorService() {
13       Log.d(TAG, "createRemoteSensorService()");
14       BluetoothGattService service = new BluetoothGattService(
15             REMOTE_SENSOR_SERVICE_UUID,
16             BluetoothGattService.SERVICE_TYPE_PRIMARY);
17
18       BluetoothGattCharacteristic characteristic = new BluetoothGattCharacteristic(
19             TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID,
20             //Read-only characteristic, supports notifications
21             BluetoothGattCharacteristic.PROPERTY_READ |
22             BluetoothGattCharacteristic.PROPERTY_NOTIFY,
23             BluetoothGattCharacteristic.PERMISSION_READ_ENCRYPTED);
24
25       //characteristic.addDescriptor(config);
26       service.addCharacteristic(characteristic);
27
28       return service;
29     }
30 }
```

Listing 4.1: RemoteSensorProfile.java

The previous characteristic has no descriptor associated. Its properties are set as `PROPERTY_NOTIFY`, means that the characteristic supports notification and `PROPERTY_READ`, means that the characteristic is readable. The permission refers only the reading: it is set with `PERMISSION_READ_ENCRYPTED`, so it is necessary an encripted link between two devices, in order to exchange the data of the characteristic (pairing is required).

On startup, the application sets the input/output capability of the PicoPi: this one has not the ability to display outputs or to receive inputs, so it will receive a `IO_CAPABILITY_NONE`.

```
mBluetoothConfigManager.setLeIoCapability(
        BluetoothConfigManager.IO_CAPABILITY_NONE);
```

After that, two Android BroadcastReceiver are registered: one to intercept whether the state of the local Bluetooth adapter has been changed, the other one to indicate a change in the bond state of the device.

```
// Register the adapter state change receiver
IntentFilter state_change_filter = new IntentFilter(
        BluetoothAdapter.ACTION_STATE_CHANGED);
registerReceiver(adapterStateChangeReceiver, state_change_filter);

// Register the device bond state change
IntentFilter bond_change_filter = new IntentFilter(
        BluetoothDevice.ACTION_BOND_STATE_CHANGED);
registerReceiver(bluetoothDeviceReceiver, bond_change_filter);
```

Then, the application enables the current Bluetooth adapter in order to make it discoverable (and available for pairing) for the next 60 seconds.

```
private static final int DISCOVERABLE_TIMEOUT_MS = 60000;
private static final int REQUEST_CODE_ENABLE_DISCOVERABLE = 100;
...
private void enableDiscoverable() {
    Log.d(TAG, "Registering for discovery.");
    Intent discoverableIntent =
        new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);

    discoverableIntent.putExtra(
        BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,
        DISCOVERABLE_TIMEOUT_MS);

    startActivityForResult(discoverableIntent,
                           REQUEST_CODE_ENABLE_DISCOVERABLE);
}
```

Next it starts the advertising process: the Bluetooth adapter name is setted up as "*PicoPiServer*" and the transmission parameters are established.

```
private void startAdvertising() {
    Log.d(TAG, "startBLEAdvertising()");

    Log.d(TAG, "Set up Bluetooth Adapter name ");
    mBluetoothAdapter.setName(ADAPTER_FRIENDLY_NAME);

    mBluetoothLeAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
    if (mBluetoothLeAdvertiser == null) {
        Log.w(TAG, "Failed to create advertiser");
        return;
    }
```

```
    AdvertiseSettings settings = new AdvertiseSettings.Builder()
      .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
      .setConnectable(true)
      .setTimeout(0) // 0 remove the time limit
      .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
      .build();

    AdvertiseData data = new AdvertiseData.Builder()
      .setIncludeDeviceName(true)
      .setIncludeTxPowerLevel(false)
      .addServiceUuid(new ParcelUuid(
            RemoteSensorProfile.REMOTE_SENSOR_SERVICE_UUID))
      .build();

    mBluetoothLeAdvertiser
            .startAdvertising(settings, data, mAdvertiseCallback);
}
```

In the end a GATT server instance is initialized, with the service and the characteristic from the RemoteSensorProfile class.

```
private void startGattServer() {
    Log.d(TAG, "startGattServer()");
    mBluetoothGattServer =
        mBluetoothManager.openGattServer(this, mGattServerCallback);
    if (mBluetoothGattServer == null) {
        Log.w(TAG, "Unable to create GATT server");
        return;
    }

    mBluetoothGattServer.addService(
        RemoteSensorProfile.createRemoteSensorService());

    handler.post(reportTemperature);
}
```

After the GATT server is initialized, the Runnable object `reportTemperature` is started, through `post()` method. We report the code below:

```
private final Runnable reportTemperature = new Runnable() {

    @Override
    public void run() {
        notifyRegisteredDevices();
        handler.postDelayed(reportTemperature, TimeUnit.SECONDS.toMillis(2));
    }
};
```

In its run method it is invoked the `notifyRegisteredDevices()` function: this latter sends a `TEMPERATURE_NOTIFY_CHARACTERISTIC` to any devices

that are subscribed to the characteristic. This operation is done using the `notifyCharacteristicChanged()` method which sends a notification or an indication to the remote device to signal that the characteristic has been updated. If there are no subscribers registered, no temperature value is sent.

```
private void notifyRegisteredDevices() {
    if (mRegisteredDevices.isEmpty()) {
        Log.i(TAG, "No subscribers registered");
        return;
    }

    pairedDevices = mBluetoothAdapter.getBondedDevices();

    // For each BT device registered
    for (BluetoothDevice device : mRegisteredDevices) {

      if(!pairedDevices.contains(device)){
            continue;
      }

      BluetoothGattCharacteristic temperatureDataCharacteristic =
        mBluetoothGattServer
        .getService(RemoteSensorProfile.
                REMOTE_SENSOR_SERVICE_UUID)
        .getCharacteristic(RemoteSensorProfile.
                TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID);

        float temp = readTemperature();
        String temperature = "";
        temperature = String.valueOf(temp);

        temperatureDataCharacteristic.setValue(temperature);

        boolean success = mBluetoothGattServer
          .notifyCharacteristicChanged(device, temperatureDataCharacteristic, false);

        Log.d(TAG, "temperature = "+temperature+" - SUCCESS = "+success);
     }
}
```

The temperature detection and the value sending are performed every 2 seconds (`handler.postDelayed(reportTemperature, TimeUnit.SECONDS .toMillis(2));`).

The management of the advertising and pairing phase and also the administration of the Gatt server makes use of specific callbacks. In particular in the code is implemented a `BluetoothPairingCallback`, necessary during the pairing process, as seen in Section 4.1.4.

```
private BluetoothPairingCallback mBluetoothPairingCallback =
    new BluetoothPairingCallback() {

    @Override
    public void onPairingInitiated(BluetoothDevice bluetoothDevice,
        PairingParams pairingParams) {
            // Handle incoming (or confirmation of outgoing) pairing request
            handlePairingRequest(bluetoothDevice, pairingParams);
        }

    @Override
    public void onPaired(BluetoothDevice bluetoothDevice) {
        // Device pairing complete
        Log.i(TAG,"Device pairing complete");
    }

    @Override
    public void onUnpaired(BluetoothDevice bluetoothDevice) {
        // Device unpaired
        Log.i(TAG, "Device unpaired");
    }

    @Override
    public void onPairingError(BluetoothDevice bluetoothDevice,
        BluetoothPairingCallback.PairingError pairingError) {
        // Something went wrong!
        Log.e(TAG,"Pairing Error "+pairingError.getErrorCode());
    }
};
```

There is also a `AdvertiseCallback`, used to deliver advertising operation status and to receive information about the advertising process.

```
private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
    @Override
    public void onStartSuccess(AdvertiseSettings settingsInEffect) {
        Log.i(TAG, "LE Advertise Started.");
    }

    @Override
    public void onStartFailure(int errorCode) {
        String error = "";
        switch (errorCode) {

            case AdvertiseCallback.ADVERTISE_FAILED_ALREADY_STARTED:
                error = "Already started";
                break;
            case AdvertiseCallback.ADVERTISE_FAILED_DATA_TOO_LARGE:
                error = "Data too large";
                break;
            case AdvertiseCallback.
                 ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
                error = "Feature unsupported";
                break;
```

```
                case AdvertiseCallback.ADVERTISE_FAILED_INTERNAL_ERROR:
                    error = "Internal error";
                    break;
                case AdvertiseCallback.
                     ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
                    error = "Too many advertisers";
                    break;
                default:
                    error = "Unknown";

                Log.e(TAG, "LE Advertise Failed: " + error);
            }
        }
};
```

The last one is the `BluetoothGattServerCallback` used to handle incoming requests to the GATT server. All the read and write requests for the characteristics and the descriptors are handled there.
Since the code of this callback is too long, we omit it, but it can be found with the entire code of this activity in the appendix A.

4.2.2 The BLEPicoClient application

The second project we have treated is called *BLEPicoClient*.

Also in this case the project consists of two applications:

- **SmartphoneServer**: the Android app which acts as a server.
- **PicoPiClient**: the Android Things app for PicoPi;

This project is inspired by an Android Things application named *ButtonThings*, available on GitHub [6] and developed by Brandon Roberts, an *Hackster.io* community member. Indeed, our two apps interacts each other in a similar way as in the ButtonThings application and they use parts of its code.
The PicoPiClient app runs in the PicoPi; the SmartphoneServer app runs in an Android smartphone (a Samsung Galaxy S8 SM-G950F runs Android 9).

The role of *Master* is played by the PicoPi: it starts scanning for a fixed Android device which advertises, in order to start a connection. This time, there is not a list of accessible devices, because the PicoPi is treated as an IoT device, thus without a display (even if a small optional LCD display is contained in the kit). When the target device is found, a connection is initialized: if the devices were already paired, the connection proceeds normally, else a pairing phase begins, with a pairing request sent by the PicoPi.

[6] `https://github.com/brandmooffin/ButtonThings/tree/part-3_remote_led`

The role of *Slave* is played by the smartphone: it broadcasts information about its service and its characteristics and waits for the incoming connection requests. After the connection is established, using a simple button on the display of the smartphone, it is possible to transimt on/off commands over the BLE channel, in order to turn on or off a LED in the PicoPi.

SmartphoneServer

The SmartphoneServer app is composed of a single activity named MainActivity and it includes the RemoteLedProfile class. As we have seen in the previuos section, where the RemoteSensorProfile was presented (4.1), this type of class represents the collection of services offered by the server.

```
1  public class RemoteLedProfile {
2
3      // Remote LED Service UUID
4      public static UUID REMOTE_LED_SERVICE =
5      UUID.fromString ("00001805-0000-1000-8000-00805f9b34fb");
6      // Remote LED Data Characteristic
7      public static UUID REMOTE_LED_DATA =
8      UUID.fromString("00002a2b-0000-1000-8000-00805f9b34fb");
9
10     private final static String TAG = RemoteLedProfile.class.getSimpleName();
11
12     // Return a configured BluetoothGattService instance for the RemoteLEDService.
13     public static BluetoothGattService createRemoteLedService() {
14         Log.d(TAG, "createRemoteLedService()");
15         BluetoothGattService service = new BluetoothGattService(
16             REMOTE_LED_SERVICE,
17             BluetoothGattService.SERVICE_TYPE_PRIMARY);
18
19         BluetoothGattCharacteristic ledData = new BluetoothGattCharacteristic(
20             REMOTE_LED_DATA,
21             //Read-only characteristic, supports notifications
22             BluetoothGattCharacteristic.PROPERTY_READ |
23             BluetoothGattCharacteristic.PROPERTY_NOTIFY,
24             BluetoothGattCharacteristic.PERMISSION_READ_ENCRYPTED);
25
26         service.addCharacteristic(ledData);
27
28         return service;
29     }
30 }
```

Listing 4.2: RemoteLedProfile.java

As it happened for the RemoteSensorProfile class, also this class implements only one service, identified by the `REMOTE_LED_SERVICE_UUID`. The service contains only a single characteristic identified by the `REMOTE_LED_DATA` UUID.

On startup, the application registers an Android BroadcastReceiver named `mBluetoothReceiver`: its role is to listen for Bluetooth adapter events in order to enable or disable the advertising and the server functionality.

```
IntentFilter filter = new IntentFilter(BluetoothAdapter.
                                        ACTION_STATE_CHANGED);
registerReceiver(mBluetoothReceiver, filter);
...
private BroadcastReceiver mBluetoothReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                                        BluetoothAdapter.STATE_OFF);

        switch (state) {
            case BluetoothAdapter.STATE_ON:
                startAdvertising();
                startServer();
                break;
            case BluetoothAdapter.STATE_OFF:
                stopServer();
                stopAdvertising();
                break;
            default:
                // Do nothing
        }
    }
};
```

After that, the application checks if the Bluetooth adapter is enabled and turns it on if necessary. Then the advertsing service and the Gatt server service are started. We report the two methods that perform these actions, `startAdvertising()` and `startServer()`:

```
private void startAdvertising() {

    Log.d(TAG, "startAdvertising()");
    BluetoothAdapter bluetoothAdapter = mBluetoothManager.getAdapter();

    // set the name
    bluetoothAdapter.setName("S8AlessandroBLE");

    mBluetoothLeAdvertiser = bluetoothAdapter.getBluetoothLeAdvertiser();

    if (mBluetoothLeAdvertiser == null) {
        Log.w(TAG, "Failed to create advertiser");
        return;
    }
```

```
    AdvertiseSettings settings = new AdvertiseSettings.Builder()
      .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
      .setConnectable(true)
      .setTimeout(0)
      .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
      .build();

    AdvertiseData data = new AdvertiseData.Builder()
      .setIncludeDeviceName(true)
      .setIncludeTxPowerLevel(false)
      .addServiceUuid(new ParcelUuid(RemoteLedProfile.
                                        REMOTE_LED_SERVICE))
      .build();

    mBluetoothLeAdvertiser
      .startAdvertising(settings, data, mAdvertiseCallback);
}
```

Before broadcasting the advertisement packets, the name of the Bluetooth adapter is set as *"S8AlessandroBLE"*: this is the name that is recognized during the scan phase by the PicoPi.

```
private void startServer() {
    Log.d(TAG, "startServer()");
    mBluetoothGattServer = mBluetoothManager.
                                openGattServer(this, mGattServerCallback);
    if (mBluetoothGattServer == null) {
        Log.w(TAG, "Unable to create GATT server");
        return;
    }

    mBluetoothGattServer.addService(RemoteLedProfile.createRemoteLedService());
}
```

The `mGattServerCallback` object, passed to the `openGattServer()` method, is an instance of BluetoothGattServerCallback class, which handles all the incoming requests to the GATT server hosted in the smartphone. All the read and write requests for the offered characteristics are handled inside it.

Since the control commands for the PicoPi LED are transmitted when the button on the smartphone display is pressed, a listener must be registered to be called when the button is clicked.

```
Button toggleButton = findViewById(R.id.toggle_button);
toggleButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        toggleLight = !toggleLight;
        notifyRegisteredDevices(toggleLight);
    }
});
```

Within the `onClick()` method it is called the `notifyRegisteredDevices( Boolean toggle)` method: this latter sends a RemoteLedService notification to any devices that are subscribed to the related characteristic.

```
private void notifyRegisteredDevices(Boolean toggle) {
    if (mRegisteredDevices.isEmpty()) {
        Log.i(TAG, "No subscribers registered");
        return;
    }

    Log.i(TAG, "Sending update to " + mRegisteredDevices.size() + " subscribers");

    for (BluetoothDevice device : mRegisteredDevices) {
        BluetoothGattCharacteristic ledDataCharacteristic = mBluetoothGattServer
                .getService(RemoteLedProfile.REMOTE_LED_SERVICE)
                .getCharacteristic(RemoteLedProfile.REMOTE_LED_DATA);
        ledDataCharacteristic.setValue(toggle.toString());
        mBluetoothGattServer.notifyCharacteristicChanged(device,
                                                         ledDataCharacteristic,
                                                         false);
    }
}
```

PicoPiClient

The PicoPiClient is the Android Things application which runs in the PicoPi device. It is composed of a single activity called *MainActivity* and it includes the *BluetoothLeService* class, the same "support" class used in the previous project by the BLEclient app.
This class implements an Android Service for managing the connection and data communication with the GATT server hosted by the Android smartphone.

On startup, the application sets the input/output capabilities of the PicoPi as `IO_CAPABILITY_NONE` (device has no input or output capabilities). Then, three Android BroadcastReceiver are registered: `mAdapterStateChangeReceiver` to handle the intents that are broadcast by the Bluetooth adapter whenever it changes its state (for example after calling `enable()`), `bluetoothDeviceReceiver` to indicate a change in the bond state of the device and

Figure 4.4: Layout of the SmartphoneServer application

`mGattUpdateReceiver` which handles various events fired by the BluetoothLeService service.

After that, it is enabled the Bluetooth adapter and it is set up its name as *"PicoPiDevice"*.

```
private static final String ADAPTER_FRIENDLY_NAME = "PicoPiDevice";
...
mBluetoothAdapter.enable();
mBluetoothAdapter.setName(ADAPTER_FRIENDLY_NAME);
```

In the end, the button "A" and the button "C" of the RainbowHAT board are configured in order to allow the devices scan process and the app closing process respectively.

```
mScanningButton = RainbowHat.openButtonA();
mScanningButton.setOnButtonEventListener(new Button.OnButtonEventListener() {
    @Override
    public void onButtonEvent(Button button, boolean pressed) {
        Log.d(TAG, "button A pressed");
        if(!mScanning) {
            enableDiscoverable(); // Enable Pairing mode (discoverable)
            scanLeDevice(); // Scan for BLE devices
        }
        else
            Log.e(TAG,"Scan already started!");
    }
});
```

```
mCloseButton = RainbowHat.openButtonC();
mCloseButton.setOnButtonEventListener(new Button.OnButtonEventListener() {
    @Override
    public void onButtonEvent(Button button, boolean pressed) {
        Log.d(TAG, "button C pressed");
        //Close the application
        MainActivity.this.finish();
    }
});
```

The `enableDiscoverable()` method, that is invoked when the button "A" is pressed, is the same method seen in Section 4.2.1 for the BLEserver application. Regarding the `scanLeDevice()` method, the only difference from the code shown in Section 4.1.4 concerns the implementation of the LeScanCallback object: this time the name of the target device to discover is *"S8AlessandroBLE"* that is the identifier for the Bluetooth adapter in the smartphone server.

```
private BluetoothAdapter.LeScanCallback mLeScanCallback =
  new BluetoothAdapter.LeScanCallback() {
    @Override
    public void onLeScan(final BluetoothDevice device, int rssi, byte[] scanRecord) {
        Boolean isDeviceFound = false;
        if(device != null){
          final String deviceName = device.getName();
          if (deviceName != null && deviceName.length() > 0) {
            Log.d(TAG, " Found device: "+deviceName);
            if(deviceName.equals(ANDROID_DEVICE_NAME)){
                isDeviceFound = true;
                mRemoteBluetoothDevice = device;
                mRemoteDeviceAddress = device.getAddress();
            }
          }
        }
        if(isDeviceFound && !mConnected) {
          Set<BluetoothDevice> pairedDevices = mBluetoothAdapter
                                                .getBondedDevices();
          if(pairedDevices.contains(mRemoteBluetoothDevice)){
            Intent gattServiceIntent = new Intent(MainActivity.this,
                                                BluetoothLeService.class);
            bindService(gattServiceIntent, mServiceConnection,
                        BIND_AUTO_CREATE);
          }
          else{
            mBluetoothConnectionManager.initiatePairing(mRemoteBluetoothDevice);
          }
        }
    }
};
```

Listing 4.3: mLeScanCallback implementation for PicoPiClient app

After the scan has been performed, if the remote target device is found and it is not already connected with the PicoPi, the application checks whether it belongs to the list of paired devices: this list is retrieved using the `getBondedDevices()` method. If the two devices were already paired, the BluetoothLeService service is started using the `bindService()` method. This service performs the connection of the PicoPi to the GATT server hosted on the smartphone. Instead, if the PicoPi and the smartphone are not paired, the pairing process begins with the call of the `initiatePairing()` method.

When the `mGattUpdateReceiver` receives an `ACTION_DATA_AVAILABLE` event, which means some data has been received from the server device, the green LED is turned on or off depending on the given value. We report only the related piece of code:

```
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "onReceive()");
        final String action = intent.getAction();

        if{
        ...
        }
        ...
        else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
          String data = intent.getStringExtra(BluetoothLeService.EXTRA_DATA);
          if (data == null)
                    return;
          if(data.contains("false")){
                setLedValue(false);
          }else if(data.contains("true")){
                setLedValue(true);
          }
        }
    }
}
```

The `setLedValue(boolean value)` method updates the output value of the green LED in the RainbowHAT board: `value` TRUE means light on, `value` FALSE means light off.

4.3 Discovering incorrect pairing

In this section, we describe some relevant bugs in the Bluetooth LE API, which we identified during the development of the previous applications. In particular the various problems regard the pairing phase between the two devices we used, PicoPi and the Smartphone. The tests we have performed,

consider the interaction between an Android smartphone and an Android Things device (PicoPi).

4.3.1 Failures in devices with `IO_CAPABILITY_NONE` which act as client

The main problem we encountered is the failure of the pairing phase when the Android Things device declares the `IO_CAPABILITY_NONE` as its input/output capability.

This bug afflicted the architecture of the BLEPicoClient project, where the PicoPi plays the role of Master and the smartphone plays the role of Slave. In the opposite architecture (BLEPicoServer project), this problem is not detected.

The following figure displays the output console of the PicoPiClient application. In particular the log reports the events captured by the `mLeScanCallback` and by the `bluetoothDeviceReceiver`.

```
D/MainActivity: Bluetooth Adapter is already enabled.
D/MainActivity: Set up Bluetooth Adapter name
D/MainActivity: button A pressed
D/MainActivity: Registering for discovery.
D/MainActivity: scanLeDevice()
D/MainActivity: Enable discoverable returned with result 120
I/MainActivity: Bluetooth adapter successfully set to discoverable mode.
D/MainActivity: found device: S8AlessandroBLE
D/MainActivity: Start pairing with S8AlessandroBLE: 5A:07:23:0E:62:C9
E/MainActivity: onPairingError 1
D/MainActivity: cancel
```

Figure 4.5: Screenshot of the log console of PicoPiClient application.

The reported error code has value equals to **1**: this value is identified in the BluetoothPairingCallback.PairingError class with the constant `UNBOND_REASON_AUTH_FAILED` and the Android Things documentation says: *A bond attempt failed because pins did not match, or remote device did not respond to pin request in time.*

To better understand the reasons about this error, and to make sure it does not depend on the current implementation of the app, we have performed the pairing process modifying the application on the PicoPi with these three strategies:

1. Set the advertising characteristic as `CARATHERISTIC_READ_ENCRYPTED` and implement the pairing phase (default behavior as we seen in Section 4.2.1).

2. Set the advertising characteristic as `CARATHERISTIC_READ`, means no encryption is necessary, and implement the pairing phase separately.

3. Set the advertising characteristic as `CARATHERISTIC_READ_ENCRYPTED` and let the Android Things OS manage the pairing process.

For each of these variants, we have observed that the pairing process fails with the same error code.

Another test we performed, concerns the use of two different methods to start the pairing phase. As seen in the description of the PicoPi-Client, in the implementation of the `mLeScanCallback` object (Listing 4.3) we adopt the `initiatePairing()` method. The other alternative we tried to use, is the `createBond()` method: it allows to start a bonding (pairing) process with a remote device and it belongs to the BluetoothDevice class (`android.bluetooth` package). We report the piece of code of the `mLeScanCallback` related to the pairing and connection phase:

```
if(isDeviceFound && !mConnected) {
  Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
  if(pairedDevices.contains(mRemoteBluetoothDevice)){
       Intent gattServiceIntent = new Intent(MainActivity.this,
                                              BluetoothLeService.class);
       bindService(gattServiceIntent, mServiceConnection, BIND_AUTO_CREATE);
   }
  else{
        mRemoteBluetoothDevice.createBond();
  }
}
```

Even in this case the pairing process fails with the error code obtained previously.

This is probably a bug in the Bluetooth LE APIs of Android Things, which afflicts the architecture where the IoT device acts as a client and it declares an I/O capability of type `IO_CAPABILITY_NONE`.

Although it may seem a non critical bug, since the Android Things BLE APIs offer other possibilities for the device capability (as seen in Section 4.1.4), it brings heavy consequences. Most IoT devices have no input peripherals and often no display, thus the only available capability for them is the `IO_CAPABILITY_NONE`. If the latter is not compatible with the pairing process but it is the only one applicable, the only choice to continue with the connection is to avoid the pairing phase.
This choice, however, involves the use of a non-encrypted connection.

4.3.2 Problems in saving pairing information

In order to analyze the behavior of the devices after the completion of the pairing process, we have changed the input/output capability of the PicoPi,

set it first as `IO_CAPABILITY_OUT` and then `IO_CAPABILITY_IN`. In this way we were able to continue the pairing phase until it was completed.

For the capability sets as `IO_CAPABILITY_OUT`, means that the device has only a display, we used the log console to simulate the display. During this test, the PicoPi device was necessarily connected to the computer running AndroidStudio. With this input/output capability, the pairing phase finished correctly, but it is necessary to insert a 6-digit pin in the server device (i.e. in the Android smartphone). The Figure 4.6 shows the log con-

```
D/MainActivity: Bluetooth Adapter is already enabled.
D/MainActivity: Set up Bluetooth Adapter name
D/MainActivity: button A pressed
D/MainActivity: Registering for discovery.
D/MainActivity: scanLeDevice()
D/MainActivity: Enable discoverable returned with result 120
I/MainActivity: Bluetooth adapter successfully set to discoverable mode.
D/MainActivity: Found device: S8AlessandroBLE
D/MainActivity: Start pairing with S8AlessandroBLE: 41:3D:60:94:DF:6E
I/MainActivity: Handle incoming pairing request or confirmation of
                outgoing pairing request
D/MainActivity: Display Passkey - 551679
I/MainActivity: Device pairing complete
```

Figure 4.6: Screenshot of the log console of PicoPiClient application during a correct pairing process.

sole of the PicoPi, where is displayed the 6-digit pin ("`Display Passkey - 551679`") and then it is invoked the `onPaired()` method belongs to the BluetoothPairingCallback, that prints the "`Device pairing complete`" string.

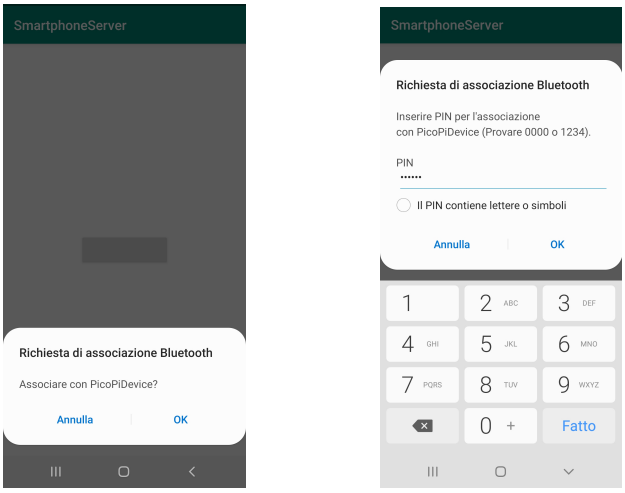
In the SmartphoneServer app, the pairing notification appears and then it is shown a dialog for digiting the pin.

Regarding the capability sets as `IO_CAPABILITY_IN`, means the device can accept only keyboard user input, we take advantage of the LCD display included in the NXP i.MX7D Starter Kit (see Section 2.1.1). This time we have developed a small layout for the MainActivity, which consists of an EditText for pin insertion and a confirmation button.

Also with this input/output capability, the pairing phase finished correctly, as happened in the previous test.

Although the pairing phase ends correctly, we have observed a critical behavior in saving pairing information.
When the PicoPiClient application is shut down and run again, e.g. forcing its termination or after rebooting the device, and the client and server try to connect again, the pairing phase starts every time.

This behavior is abnormal since the pairing information is saved correctly on both devices: in the smartphone the association with the *"PicoPiDevice"* is visible in the typical Bluetooth menu of the Android device, instead in the Android Things device it is possible to obtain the list of the paired devices



(a) Pairing notification (b) Pin dialog

Figure 4.7: SmartphoneServer application during the pairing process.

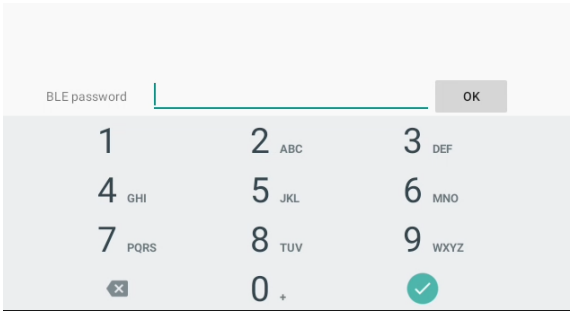


Figure 4.8: Layout for the PicoPiClient application.

using the `getBondedDevices()` method, belongs to the BluetoothAdapter class (in this case the list is visible in the log console).
This is a simple method used to show the list of bonded devices with the PicoPi:

```
private void listBondedDevices(){
    pairedDevices = mBluetoothAdapter.getBondedDevices();
    if (pairedDevices.size() > 0) {
        Log.d(TAG, "|Bondend Devices|");
        for (BluetoothDevice device: pairedDevices){
            Log.d(TAG, "|Device = "+device.getName()+
                        " (address = "+device.getAddress()+") |");
        }
    }
}
```

```
D/MainActivity: Bluetooth Adapter is already enabled.
D/MainActivity: Set up Bluetooth Adapter name
D/MainActivity: button A pressed
D/MainActivity: Registering for discovery.
D/MainActivity: scanLeDevice()
D/MainActivity: Enable discoverable returned with result 120
I/MainActivity: Bluetooth adapter successfully set to discoverable mode.
D/MainActivity: Found device: S8AlessandroBLE
D/MainActivity: Start pairing with S8AlessandroBLE: 59:EF:AF:35:46:33
I/MainActivity: Handle incoming pairing request or confirmation of
                outgoing pairing request
D/MainActivity: pin = 334037
I/MainActivity: Device pairing complete
```

(a) PicoPiClient log console

(b) Pin dialog in SmartphoneServer app

Figure 4.9: SmartphoneServer and PicoPiClient application during the pairing process.

In this implementation, the pairing process starts into the `mLeScanCallback` object, where before invoking the `initiatePairing()` method it is manually checked whether the target device, found during the scanning, is contained in the bondend devices list.
Using the Android Studio console, we have observed that this check returns `false` every time: this means the target device, that is the Android smartphone that acts as a server, is considered as a new device with each new connection.

As can be seen in Figure 4.10, the list of devices paired with PicoPi includes several times the same Android device, the *S8AlessandroBLE*, which has different MAC addresses associated: this indicates that the saved pairing information is not used in subsequent connections.

```
D/MainActivity: |Bondend Devices|
D/MainActivity: |Device = S8AlessandroBLE (address = 7C:53:41:78:DC:C1) |
D/MainActivity: |Device = S8AlessandroBLE (address = 6A:5A:8D:E8:0C:4C) |
D/MainActivity: |Device = S8AlessandroBLE (address = 59:27:61:2B:C7:1E) |
D/MainActivity: |Device = S8AlessandroBLE (address = 7F:40:69:FD:24:67) |
```

Figure 4.10: The list of bonded device with the *S8AlessandroBLE* device saved several times.

On the "Slave" side, thus on the Android smartphone, the pairing information remains valid for a certain time interval (about some minutes). During this period, at each pairing request from the client, the related information previously saved it is not discarded and treated as valid.
After this period, when a pairing request arrives, new pairing data are saved in the smartphone.

We test the interaction of the two devices, changing the pairing strategies, as we performed for the previous case described in Section 4.3.1. The description of the previous problems refers to the default “strategy”, where we used the advertising characteristic as `CARATHERISTIC_READ_ENCRYPTED` and we implemented the pairing phase. Afterwards, we set the advertising characteristic as `CARATHERISTIC_READ`, and we performed the pairing phase separately. In the end, we tried a third way: we set the advertising characteristic as `CARATHERISTIC_READ_ENCRYPTED` and let the Android Things OS manage the pairing process.
The behavior of the two devices is the same for all the three situations, with the same problems discussed above.

This problem in saving the pairing data and in their subsequent use, seems to be associated to the inability of the Android Things devices, that act as a client, to resolve the various random MAC addresses of the server. As mentioned in Section 4.1.3, this is a BLE specification that allows to prevent device tracking.

This bug implies the presence of two critical aspects:

- First of all, if the pairing is required every time, an attacker who sniffs the initial phase of the connection, is able to intercept the pairing information, even if a pairing phase has been performed previously. This is a contradiction in the usefulness of the pairing, whose goal is to protect the succeeding connections.

- The second important aspect concerns the failure of the subsequent pairing phases: as we noted in our tests if the first process ends successfully, but the second fails, the pairing data previously saved are canceled. Indeed, after successfully completing the association between the PicoPi and the smartphone, we forced the second pairing process

to fail. The failure caused the deletion of the pairing information correctly saved on the smartphone and relating to the PicoPi.
An attacker could force the pairing process to fail, in order to delete the previously saved information.

4.3.3 Minor bugs

A minor bug we found concerns the Android Things device that acts as server, thus we refer to the BLEPicoServer project (4.2.1) and its architecture where the PicoPi makes the advertisement and waits for incoming connection request.

Following the official Android Things documentation, we were not able to activate the advertising process. In fact, the `onStartSuccess()` method, contained in the AdvertiseCallback of the BLEserver app, has never been invoked, a sign that the advertisement has not been started. Also in the BLEclient application, none devices appeared in the menu of available devices.

The problem has been solved by deleting the line of code where the `addServiceUuid()` method is called, within the creation of the AdvertiseData object, in the `startAdvertising()` method of the BLEserver app.

```
AdvertiseData data = new AdvertiseData.Builder()
    .setIncludeDeviceName(true)
    .setIncludeTxPowerLevel(false)
    //.addServiceUuid(new ParcelUuid(
        //RemoteSensorProfile.REMOTE_SENSOR_SERVICE_UUID))
    .build();
```

After this adjustment the advertising process starts correctly.

Chapter 5

Conclusions and future work

IoT technology plays an increasingly important role in everyday life, with a pervasive presence on the one hand, but essential on the other. Because of its wide range of applications, it is extremely important to understand the risks and security challenges involved.

Understanding and analyzing the Android Things platform, developed by Google which is one of the most significant IT players in the market, is a first important step.

On the one hand, with this project it was possible to study how the Android Things APIs interact with the various sensors and the various peripherals (Gpio, I2C and PWM) and subsequently present a way of developing the IoT software, closest to the development of mobile applications, thanks to the use of the Android Architecture Components. Unlike the approach present in the various projects available online, characterized by the use of a single activity, it is possible to achieve greater modularity and portability of the code, essential characteristics when the projects increase their size and they integrate new functionalities.

On the other hand, through the development of applications that use the Bluetooth Low Energy technology, it was possible to detect some relevant vulnerabilities of the Android Things Bluetooth LE API.

An interesting future work is to analyze the other Android Things APIs concerning wireless communication, for example the WiFi API and the 6LoWPAN API, in order to detect possible security vulnerabilities.

Although Google recently announced that the Android Things OS will be primarily aimed to OEM partners rather than developers, we believe that most of the APIs of the Android Things Support Library will be the core of the applications developed by these partners. Therefore, a careful analysis of the various bugs and vulnerabilities of this platform plays an important role for its future development, whatever it is.

Appendix A

Android Things applications code

A.1 PicoTemperature_GodActivity

MainActivity

```
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.getSimpleName();
    // Constants for different temperature threshold
    private static final float MAX_TEMPERATURE = 28.0f;
    private static final float NORMAL_TEMPERATURE = 22.0f;
    // Constant for brightness max value
    public static final int HT16K33_BRIGHTNESS_MAX = 0b00001111;
    // Variables for RainbowHAT peripherals
    private Bmx280 tempSensor;
    private AlphanumericDisplay mDisplay;
    private Speaker buzzer;
    private Button buttonC;
    private Gpio ledR, ledG, ledB;
    private Handler myHandler;
    private boolean playing = false;

    private void initButton() {
        try {
          buttonC = RainbowHat.openButtonC();
          buttonC.setOnButtonEventListener(new Button.OnButtonEventListener() {
                @Override
                public void onButtonEvent(Button button, boolean pressed) {
                    Log.d(TAG, "button C pressed");
                    MainActivity.this.finish();
                }
          });
        }
        catch (IOException e){
            Log.e(TAG, "Unable to open Button C");
        }
    }
```

```
33
34      public void setLedLight(char led, boolean value){
35          switch (led){
36              case 'R':
37                  try {
38                      ledR.setValue(value);
39                      break;
40                  }
41                  catch (IOException e){
42                      Log.e(TAG, "Unable to manage RED led");
43                  }
44              case 'B':
45                  try {
46                      ledB.setValue(value);
47                      break;
48                  }
49                  catch (IOException e){
50                      Log.e(TAG, "Unable to manage GREEN led");
51                  }
52              case 'G':
53                  try {
54                      ledG.setValue(value);
55                      break;
56                  }
57                  catch (IOException e){
58                      Log.e(TAG, "Unable to manage BLUE led");
59                  }
60              default:
61                  Log.e(TAG, "Invalid led identifier. Allowed values: R,G,B");
62          }
63      }
64
65
66      private final Runnable reportTemperature = new Runnable() {
67
68          float temperature;
69
70          @Override
71          public void run() {
72              if(playing) {
73                  try {
74                      // Stop the buzzer.
75                      buzzer.stop();
76                      playing = false;
77                  }
78                  catch (IOException e){}
79              }
80              try {
81                  temperature = tempSensor.readTemperature();
82                  BigDecimal tempBG = new BigDecimal(temperature);
83                  tempBG = tempBG.setScale(2, BigDecimal.ROUND_HALF_UP);
84                  temperature = (tempBG.floatValue());
85              }
86              catch (IOException | IllegalStateException e){
87                  Log.e(TAG, "Unable to read temperature");
88                  temperature = Float.valueOf(null);
89              }
```

```
90
91              // Display temperature value and turn on the correct light
92              try {
93                  if(temperature < NORMAL_TEMPERATURE){
94                      setLedLight('R',false);
95                      setLedLight('G',false);
96                      setLedLight('B',true);
97                  }
98                  else if(temperature >= NORMAL_TEMPERATURE &&
99                          temperature < MAX_TEMPERATURE){
100                     setLedLight('R',false);
101                     setLedLight('G',true);
102                     setLedLight('B',false);
103                 }
104                 else {
105                     setLedLight('R',true);
106                     setLedLight('G',false);
107                     setLedLight('B',false);
108
109                     // Play a note on the piezo buzzer at 2000 Hz for 2 seconds
110                     try {
111                         buzzer.play(2000);
112                         playing = true;
113                     }
114                     catch (IOException e){
115                         Log.e(TAG, "Unable to play a sound with the piezo buzzer");
116                     }
117                 }
118
119                 if (temperature != null)
120                     temperature_string = String.valueOf(temperature);
121                 else
122                     temperature_string = "--"
123
124                 mDisplay.display(temperature_string);
125                 Log.d(TAG, "temperatura: " + temperature + "C");
126             }
127             catch (IOException e){
128                 Log.e(TAG, "Unable to write in mDisplay display");
129             }
130             catch (InterruptedException e){
131                 Log.e(TAG, "Unable to write in mDisplay display");
132             }
133
134             myHandler.postDelayed(reportTemperature,
135                                   TimeUnit.SECONDS.toMillis(2));
136         }
137     };
```

```
138
139     @Override
140     protected void onCreate(Bundle savedInstanceState) {
141         super.onCreate(savedInstanceState);
142         Log.d(TAG, "onCreate");
143
144
145         // Open led connection
146         try {
147             ledR = RainbowHat.openLedRed();
148             ledG = RainbowHat.openLedGreen();
149             ledB = RainbowHat.openLedBlue();
150         }
151         catch (IOException e){
152             Log.e(TAG, "Unable to open leds");
153         }
154
155         // Turn off the led lights, opened by default
156         setLedLight('R',false);
157         setLedLight('G',false);
158         setLedLight('B',false);
159
160         // Open temperature sensor
161         try {
162           tempSensor = RainbowHat.openSensor();
163           tempSensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
164           tempSensor.setMode(Bmx280.MODE_NORMAL);
165         }
166         catch (IOException e){
167           Log.e(TAG, "Unable to open BMP280 sensor");
168         }
169
170         // Open segment display
171         try {
172             mDisplay = RainbowHat.openDisplay();
173             mDisplay.setBrightness(HT16K33_BRIGHTNESS_MAX);
174             mDisplay.setEnabled(true);
175         }
176         catch (IOException e){
177             Log.e(TAG, "Unable to open segment display");
178         }
179
180         // Open speaker
181         try {
182             buzzer = RainbowHat.openPiezo();
183         }
184         catch (IOException e){
185             Log.e(TAG, "Unable to open the speaker");
186         }
187
188         // Initialize the button C for close the app
189         initButton();
190         // Initialize the handler
191         myHandler = new Handler(Looper.getMainLooper());
192     }
193
```

```
194
195     @Override
196     protected void onStart() {
197         super.onStart();
198         // Start temperature detection
199         myHandler.post(reportTemperature);
200     }
201
202     @Override
203     protected void onStop() {
204         Log.d(TAG, "onStop called.");
205
206         // Stop temperature detection
207         myHandler.removeCallbacks(reportTemperature);
208
209         try {
210             buttonC.close();
211         }
212         catch (IOException e){
213             Log.e(TAG, "Unable to close Button C");
214         }
215
216         // Turn off the led lights, opened by default
217         setLedLight('R',false);
218         setLedLight('G',false);
219         setLedLight('B',false);
220
221
222         // Close led connection
223         try {
224             ledR.close();
225             ledG.close();
226             ledB.close();
227         }
228         catch (IOException e){
229             Log.e(TAG, "Unable to close leds");
230         }
231
232         // Close temperature sensor
233         try {
234             tempSensor.close();
235         }
236         catch (IOException e){
237             Log.e(TAG, "Unable to close BMP280 sensor");
238         }
239
240
241         // Close segment display
242         try {
243             mDisplay.clear();
244             mDisplay.setEnabled(false);
245             mDisplay.close();
246         }
247         catch (IOException e){
248             Log.e(TAG, "Unable to close segment display");
249         }
250
```

```
251
252             // Close speaker
253             try {
254                 if(playing){
255                     buzzer.stop();
256                 }
257                 buzzer.close();
258             }
259
260             catch (IOException e){
261                 Log.e(TAG, "Unable to close the speaker");
262             }
263             super.onStop();
264         }
265
266         @Override
267         protected void onDestroy() {
268             Log.d(TAG, "onDestroy");
269             super.onDestroy();
270         }
271     }
```

Listing A.1: MainActivity.java

A.2 PicoTemperature_AAC

MainActivity

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = MainActivity.class.getSimpleName();

    // Constants for different temperature threshold
    private static final float MAX_TEMPERATURE = 28.0f;
    private static final float NORMAL_TEMPERATURE = 22.0f;
    // Constant for I2C bus on RainbowHat
    private static final String DEFAULT_I2C_BUS = "I2C1";

    private final PeripheralManager mPeripheralManager =
    PeripheralManager.getInstance();

    private MainActivityViewModel mainActivityViewModel;

    private final Observer<Boolean> exitButtonLiveDataObserver =
    new Observer<Boolean>(){
        @Override
        public void onChanged(@Nullable Boolean pressed){
            if (pressed){
                Log.d(TAG, "onChanged() in exitButtonLiveDataObserver);
                MainActivity.this.finish();
            }
        }
    };

    private final Observer<Float> temperatureLiveDataObserver =
    new Observer<Float>() {
        @Override
        public void onChanged(@Nullable Float temperature) {

            Log.d(TAG, "onChanged() in temperatureLiveDataObserver);
            mainActivityViewModel.display(temperature);
            if(temperature < NORMAL_TEMPERATURE){
                mainActivityViewModel.setLedLight('R',false);
                mainActivityViewModel.setLedLight('G',false);
                mainActivityViewModel.setLedLight('B',true);
            }
            else if(temperature >= NORMAL_TEMPERATURE &&
                    temperature < MAX_TEMPERATURE){
                mainActivityViewModel.setLedLight('R',false);
                mainActivityViewModel.setLedLight('G',true);
                mainActivityViewModel.setLedLight('B',false);
            }
            else {
                mainActivityViewModel.setLedLight('R',true);
                mainActivityViewModel.setLedLight('G',false);
                mainActivityViewModel.setLedLight('B',false);

                mainActivityViewModel.playSound();
            }
        }
    };
```

```
54
55      @Override
56      protected void onCreate(Bundle savedInstanceState) {
57          super.onCreate(savedInstanceState);
58          Log.d(TAG, "onCreate");
59
60          mainActivityViewModel = ViewModelProviders.of(this)
61                                          .get(MainActivityViewModel.class);
62
63          // Start observing ButtonLiveData
64          mainActivityViewModel.getButtonLiveData()
65          .observe(MainActivity.this, exitButtonLiveDataObserver);
66          // Start observing TemperatureLiveData
67          mainActivityViewModel.getTemperatureLiveData()
68          .observe(MainActivity.this, temperatureLiveDataObserver);
69      }
70
71      @Override
72      protected void onDestroy() {
73          super.onDestroy();
74          Log.d(TAG, "onDestroy");
75      }
76  }
```

Listing A.2: MainActivity.java

MainActivityViewModel

```
1   public class MainActivityViewModel extends ViewModel {
2
3       private static final String TAG = MainActivityViewModel.class.getSimpleName();
4
5       private final ButtonLiveData mButtonLiveData;
6       private final TemperatureLiveData mTemperatureLiveData;
7       private final AlphanumericDisplay alphanumericDisplay;
8       private final Speaker mSpeaker;
9       private final Gpio ledR, ledG, ledB;
10
11      public MainActivityViewModel(){
12          Log.d(TAG, "MainActivityViewModel instance created");
13
14          // Create LiveData
15          this.mButtonLiveData = new ButtonLiveData();
16          this.mTemperatureLiveData = new TemperatureLiveData();
17          // Open a connection to the leds
18          openLeds();
19          // Turn off leds light when init the ViewModel
20          setLedLight('R',false);
21          setLedLight('G',false);
22          setLedLight('B',false);
23          // Open a connection to the display
24          openDisplay();
25          // Open a connection to the speaker
26          openSpeaker();
27      }
```

```
28
29      public LiveData<Boolean> getButtonLiveData() {
30          return mButtonLiveData;
31      }
32
33      public LiveData<Float> getTemperatureLiveData(){
34          return mTemperatureLiveData;
35      }
36
37      private void openLeds(){
38          try{
39              ledR = RainbowHat.openLedRed();
40              ledB = RainbowHat.openLedBlue();
41              ledG = RainbowHat.openLedGreen();
42              Log.d(TAG, "Open LEDs connection");
43          }
44          catch (IOException e) {
45              Log.e(TAG, "Unable to open leds connection");
46          }
47      }
48
49      public void setLedLight(char led, boolean value){
50
51          switch (led){
52              case 'R':
53                  try {
54                      ledR.setValue(value);
55                      break;
56                  }
57                  catch (IOException e){
58                      Log.e(TAG, "Unable to manage the led"+String.valueOf(led));
59                  }
60
61              case 'B':
62                  try {
63                      ledB.setValue(value);
64                      break;
65                  }
66                  catch (IOException e){
67                      Log.e(TAG, "Unable to manage the led"+String.valueOf(led));
68                  }
69
70              case 'G':
71                  try {
72                      ledG.setValue(value);
73                      break;
74                  }
75                  catch (IOException e){
76                      Log.e(TAG, "Unable to manage the led"+String.valueOf(led));
77                  }
78
79              default:
80                  Log.e(TAG, "Invalid led identifier. Allowed values: R,G,B");
81          }
82      }
83
84
85
```

```
86
87      private void closeLeds(){
88
89          setLedLight('R',false);
90          setLedLight('G',false);
91          setLedLight('B',false);
92
93          try {
94              ledR.close();
95              ledG.close();
96              ledB.close();
97              Log.d(TAG, "Close LEDs connection");
98          }
99          catch (IOException e) {
100             Log.e(TAG, "Unable to close leds connection");
101         }
102     }
103
104
105     private void openSpeaker(){
106         if(mSpeaker == null){
107             try {
108                 mSpeaker = RainbowHat.openPiezo();
109                 Log.d(TAG, "Open speaker connection");
110             }
111             catch (IOException e){
112                 Log.e(TAG, "Unable to open the Speaker");
113             }
114         }
115     }
116
117
118     private void closeSpeaker(){
119         if(mSpeaker == null)
120             return;
121         try {
122             mSpeaker.close();
123             Log.d(TAG, "Close speaker connection");
124         }
125         catch (IOException e){
126             Log.e(TAG, "Unable to close the Speaker");
127         }
128     }
129
130
131     public void playSound(){
132         if(mSpeaker == null){
133             this.openSpeaker();
134         }
135         // An Handler for buzzer sound.
136         // It is an asynchronous thread but still on the main thread
137         Handler buzzerSoundHandler = new Handler(Looper.getMainLooper());
138         buzzerSoundHandler.post(playSound);
139     }
140
141
142
```

```
143
144     private final Runnable playSound = new Runnable() {
145         @Override
146         public void run() {
147             try {
148                 mSpeaker.play(2000);
149                 Thread.sleep(1500);
150                 mSpeaker.stop();
151             } catch (IOException | InterruptedException | IllegalStateException e) {
152                 Log.e(TAG,"Unable to play buzzer sound: "+e.toString());
153             }
154         }
155     };
156
157     private void openDisplay(){
158         if (alphanumericDisplay == null) {
159             try {
160               alphanumericDisplay = RainbowHat.openDisplay();
161               alphanumericDisplay.setBrightness(Ht16k33
162                                                 .HT16K33_BRIGHTNESS_MAX);
163               alphanumericDisplay.clear();
164               alphanumericDisplay.setEnabled(true);
165               Log.d(TAG, "Open display connection");
166             }
167             catch (IOException e) {
168               Log.d(TAG, "display: " + e);
169               alphanumericDisplay = null;
170               return;
171             }
172         }
173         else{
174             Log.d(TAG, "The display is already opened");
175         }
176     }
177
178     public void display(Float value) {
179         try {
180             alphanumericDisplay.display(value);
181         }
182         catch (IOException e) {
183             Log.d(TAG, "display: " + e);
184         }
185     }
186
187     private void closeDisplay(){
188         if (alphanumericDisplay != null) {
189             try {
190                 alphanumericDisplay.clear();
191                 alphanumericDisplay.setEnabled(true);
192                 alphanumericDisplay.close();
193                 Log.d(TAG, "Close display connection");
194             } catch (IOException e) {
195                 Log.e(TAG,"Unable to close the display);
196             }
197         }
198     }
```

```
199
200     @Override
201     protected void onCleared() {
202         // Close LEDs light and the connections
203         closeLeds();
204         // Close speaker connection
205         closeSpeaker();
206         // Clean the display and close the connection
207         closeDisplay();
208     }
209 }
```

Listing A.3: MainActivityViewModel.java

ButtonLiveData

```
1  public class ButtonLiveData extends LiveData<Boolean>{
2
3      private static final String TAG = ButtonLiveData.class.getSimpleName();
4      private Button buttonC;
5
6      @Override
7      protected void onActive() {
8          super.onActive();
9          Log.d(TAG, "onActive");
10         try {
11             buttonC = RainbowHat.openButtonC();
12             buttonC.setOnButtonEventListener(new Button.OnButtonEventListener() {
13                 @Override
14                 public void onButtonEvent( Button button, boolean pressed) {
15                     Log.d(TAG, "button C pressed");
16                     setValue(true);
17                 }
18             });
19         }
20         catch (IOException e){
21             Log.e(TAG, "Unable to open button C");
22         }
23     }
24
25     @Override
26     protected void onInactive() {
27         try {
28             buttonC.close();
29         }
30         catch (IOException e){
31             Log.e(TAG, "OnInactive: "+e);
32         }
33         super.onInactive();
34         Log.d(TAG, "onInactive");
35     }
36 }
```

Listing A.4: ButtonLiveData.java

TemperatureLiveData

```
public class TemperatureLiveData extends LiveData<Float>{

    private static final String TAG = TemperatureLiveData.class.getSimpleName();
    private Bmx280 tempSensor;
    private Handler handler;

    private final Runnable reportTemperature = new Runnable() {
        float temperature;
        @Override
        public void run() {
          try {
              temperature = tempSensor.readTemperature();
              BigDecimal tempBG = new BigDecimal(temperature);
              tempBG = tempBG.setScale(2, BigDecimal.ROUND_HALF_UP);
              temperature = (tempBG.floatValue());
              setValue(temperature);
          }
          catch (IOException | IllegalStateException e){
              Log.e(TAG, "Unable to read temperature");
              temperature = Float.valueOf(null);
          }
          handler.postDelayed(reportTemperature, TimeUnit.SECONDS.toMillis(2));
        }
    };
    @Override
    protected void onActive() {
        super.onActive();
        Log.d(TAG, "onActive");
        try {
          tempSensor = RainbowHat.openSensor();
          tempSensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
          tempSensor.setMode(Bmx280.MODE_NORMAL);
        }
        catch (IOException e){
            Log.e(TAG, "Unable to open temperature sensor");
        }
        handler = new Handler(Looper.getMainLooper());
        handler.post(reportTemperature);
    }
    @Override
    protected void onInactive() {
        handler.removeCallbacks(reportTemperature);
        try {
            tempSensor.close();
        }
        catch (IOException e) {
        }
        super.onInactive();
        Log.d(TAG, "onInactive");
    }
}
```

Listing A.5: TemperatureLiveData.java

A.3 PicoPiComponentTest

MainActivity

```
1  public class MainActivity extends Activity {
2
3      private static final String TAG = MainActivity.class.getSimpleName();
4
5      // String which identifies the I2C bus on PicoPiIMX7D
6      private static final String DEFAULT_I2C_BUS = "I2C1";
7      // slave address value for Bmx280 sensor (default)
8      private static final int DEFAULT_I2C_ADDRESS = 0x77;
9
10     private I2cDevice mI2cDevice;
11
12     @Override
13     protected void onCreate(Bundle savedInstanceState) {
14         super.onCreate(savedInstanceState);
15         Log.d(TAG, "onCreate");
16
17         // Opening a connection with an I2C component
18         try {
19           PeripheralManager mPeripheralManager = PeripheralManager.getInstance();
20           mI2cDevice = mPeripheralManager
21           .openI2cDevice(DEFAULT_I2C_BUS, DEFAULT_I2C_ADDRESS);
22           // dummy usage of mI2cDevice object
23           mI2cDevice.getName();
24         } catch (IOException e){
25             Log.e(TAG, "Unable to access I2C device", e);
26         }
27         Log.d(TAG, "Invoke MainActivity.this.finish()");
28         MainActivity.this.finish();
29     }
30
31     @Override
32     protected void onDestroy() {
33
34         if (mI2cDevice != null) {
35             try {
36                 mI2cDevice.close();
37             } catch (IOException e) {
38                 Log.w(TAG, "Unable to close I2C device", e);
39             }
40         }
41         Log.d(TAG, "onDestroy");
42         super.onDestroy();
43     }
44 }
```

Listing A.6: MainActivity.java

A.4 BLEPicoServer

A.4.1 BLEserver

```
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.getSimpleName();

    private static final String ADAPTER_FRIENDLY_NAME = "PicoPiServer";
    private static final int DISCOVERABLE_TIMEOUT_MS = 60000;
    private static final int REQUEST_CODE_ENABLE_DISCOVERABLE = 100;

    // BluetoothManager: used to determine if bluetooth is enabled and available
    private BluetoothManager mBluetoothManager;
    // BluetoothConfigManager: used to set the I/O capability of the AT device
    private BluetoothConfigManager mBluetoothConfigManager;
    // BluetoothGattServer: used to configure the BT server
    private BluetoothGattServer mBluetoothGattServer;
    // BluetoothAdapter: used to manage the BT receiver
    private BluetoothAdapter mBluetoothAdapter;
    // BluetoothLeAdvertiser: used for transmission
    private BluetoothLeAdvertiser mBluetoothLeAdvertiser;

    private BluetoothConnectionManager mBluetoothConnectionManager;

    // Set for containing the connected devices
    private Set<BluetoothDevice> mRegisteredDevices = new HashSet<>();
    // Set for containing the paired devices
    private Set<BluetoothDevice> pairedDevices = null;
    // temperature sensor
    private Bmx280 tempSensor;

    private Handler handler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mBluetoothManager = (BluetoothManager) getSystemService(
                                                BLUETOOTH_SERVICE);

        mBluetoothConfigManager = BluetoothConfigManager.getInstance();

        mBluetoothAdapter = mBluetoothManager.getAdapter();

        mBluetoothConnectionManager = BluetoothConnectionManager.getInstance();

        // Parameters for the I/O capability of the device
        mBluetoothConfigManager
            .setLeIoCapability(BluetoothConfigManager.IO_CAPABILITY_NONE);

        // Register the adapter state change receiver
        IntentFilter state_change_filter = new IntentFilter(
            BluetoothAdapter.ACTION_STATE_CHANGED);
        registerReceiver(adapterStateChangeReceiver, state_change_filter);


```

```
56
57          // Register the device bond state change
58          IntentFilter bond_change_filter = new IntentFilter(
59                  BluetoothDevice.ACTION_BOND_STATE_CHANGED);
60          registerReceiver(bluetoothDeviceReceiver, bond_change_filter);
61          // Register pairing callback
62          mBluetoothConnectionManager
63                  .registerPairingCallback(mBluetoothPairingCallback);
64
65
66          // Init the handler
67          handler = new Handler(Looper.getMainLooper());
68
69          // Open a connection to the temperature sensor
70          try {
71                  tempSensor = openTempSensor();
72          }
73          catch (IOException e){
74                  Log.e(TAG, "Unable to open temperatre sensor!");
75                  MainActivity.this.finish();
76          }
77
78          if (!mBluetoothAdapter.isEnabled()) {
79                  Log.d(TAG, "Bluetooth is currently disabled...enabling");
80                  mBluetoothAdapter.enable();
81          }else {
82                  Log.d(TAG, "Bluetooth enabled...starting services");
83                  enableDiscoverable();
84                  startAdvertising();
85                  startGattServer();
86          }
87      }
88
89      @Override
90      protected void onDestroy(){
91          super.onDestroy();
92
93          handler.removeCallbacks(reportTemperature);
94          if(tempSensor != null){
95              try {
96                  closeTempSensor();
97              }
98              catch (IOException e){
99                  Log.e(TAG, "Unable to close temperature sensor");
100             }
101         }
102         if (mBluetoothAdapter.isEnabled()) {
103             stopGattServer();
104             stopAdvertising();
105         }
106         unregisterReceiver(adapterStateChangeReceiver);
107         unregisterReceiver(bluetoothDeviceReceiver);
108         mBluetoothConnectionManager
109             .unregisterPairingCallback(mBluetoothPairingCallback);
110     }
111
112
```

```
113
114     /*
115      * Listens for Bluetooth adapter events to enable/disable
116      * advertising and server functionality.
117      */
118     private BroadcastReceiver adapterStateChangeReceiver = new BroadcastReceiver() {
119         @Override
120         public void onReceive(Context context, Intent intent) {
121             Log.d(TAG, "onRecive for action: "+intent.getAction());
122
123             int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
124                                            BluetoothAdapter.STATE_OFF);
125             switch (state) {
126                 case BluetoothAdapter.STATE_ON:
127                     Log.i(TAG, "BT Adapter is on");
128                     listBondedDevices();
129                     startAdvertising();
130                     startGattServer();
131                     break;
132                 case BluetoothAdapter.STATE_OFF:
133                     stopGattServer();
134                     stopAdvertising();
135                     break;
136                 default:
137                     // Do nothing
138             }
139         }
140     };
141     private BroadcastReceiver bluetoothDeviceReceiver = new BroadcastReceiver() {
142         @Override
143         public void onReceive(Context context, Intent intent) {
144             if( BluetoothDevice.ACTION_BOND_STATE_CHANGED
145             .equals( intent.getAction() ) ) {
146
147                 BluetoothDevice device =
148                 intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
149                 switch (device.getBondState()) {
150                     case BluetoothDevice.BOND_BONDING:
151                         Log.d(TAG, "start pairing with "+ device.getAddress());
152                         break;
153                     case BluetoothDevice.BOND_BONDED:
154                         Log.d(TAG, "finish pairing with "+device.getAddress());
155                         break;
156                     case BluetoothDevice.BOND_NONE:
157                         Log.d(TAG, "cancel");
158                         break;
159                     case BluetoothDevice.ERROR:
160                         Log.d(TAG, "error");
161                     default:
162                         break;
163                 }
164             }
165             if( BluetoothDevice.ACTION_PAIRING_REQUEST
166             .equals( intent.getAction())){
167                 Log.e(TAG, "ACTION_PAIRING_REQUEST");
168             }
169         }
170     };
```

```
171
172     /*
173      * Begin advertising over Bluetooth and supports the Service.
174      */
175     private void startAdvertising() {
176         Log.d(TAG, "startBLEAdvertising()");
177         Log.d(TAG, "Set up Bluetooth Adapter name ");
178         mBluetoothAdapter.setName(ADAPTER_FRIENDLY_NAME);
179         mBluetoothLeAdvertiser = mBluetoothAdapter.getBluetoothLeAdvertiser();
180         if (mBluetoothLeAdvertiser == null) {
181             Log.w(TAG, "Failed to create advertiser");
182             return;
183         }
184
185         AdvertiseSettings settings = new AdvertiseSettings.Builder()
186           .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
187           .setConnectable(true)
188           .setTimeout(0) // 0 remove the time limit
189           .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
190           .build();
191         AdvertiseData data = new AdvertiseData.Builder()
192             .setIncludeDeviceName(true)
193             .setIncludeTxPowerLevel(false)
194             .build();
195
196         mBluetoothLeAdvertiser
197             .startAdvertising(settings, data, mAdvertiseCallback);
198     }
199
200     /*
201      * Stop Bluetooth advertisements.
202      */
203     private void stopAdvertising() {
204         Log.i(TAG, "Stop BLE Advertising");
205         if (mBluetoothLeAdvertiser == null) return;
206
207         mBluetoothLeAdvertiser.stopAdvertising(mAdvertiseCallback);
208     }
209
210     /*
211      * Initialize the GATT server instance with the services/characteristics
212      * from the Remote LED Profile.
213      */
214     private void startGattServer() {
215         Log.d(TAG, "startGattServer()");
216         mBluetoothGattServer = mBluetoothManager
217                 .openGattServer(this, mGattServerCallback);
218         if (mBluetoothGattServer == null) {
219             Log.w(TAG, "Unable to create GATT server");
220             return;
221         }
222         mBluetoothGattServer
223                 .addService(RemoteSensorProfile.createRemoteSensorService());
224         handler.post(reportTemperature);
225     }
226
227
228
```

```
229
230     /*
231      * Shut down the GATT server.
232      */
233     private void stopGattServer() {
234         Log.d(TAG, "StopGattServer()");
235         if (mBluetoothGattServer == null) return;
236
237         mBluetoothGattServer.close();
238     }
239
240     /*
241      * Callback to receive information about the advertisement process.
242      */
243     private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
244
245         @Override
246         public void onStartSuccess(AdvertiseSettings settingsInEffect) {
247             Log.i(TAG, "LE Advertise Started.");
248         }
249
250         @Override
251         public void onStartFailure(int errorCode) {
252             String error = "";
253             switch (errorCode) {
254                 case AdvertiseCallback
255                     .ADVERTISE_FAILED_ALREADY_STARTED:
256                     error = "Already started";
257                     break;
258                 case AdvertiseCallback
259                     .ADVERTISE_FAILED_DATA_TOO_LARGE:
260                     error = "Data too large";
261                     break;
262                 case AdvertiseCallback
263                     .ADVERTISE_FAILED_FEATURE_UNSUPPORTED:
264                     error = "Feature unsupported";
265                     break;
266                 case AdvertiseCallback
267                     .ADVERTISE_FAILED_INTERNAL_ERROR:
268                     error = "Internal error";
269                     break;
270                 case AdvertiseCallback
271                     .ADVERTISE_FAILED_TOO_MANY_ADVERTISERS:
272                     error = "Too many advertisers";
273                     break;
274                 default:
275                     error = "Unknown";
276
277                 Log.e(TAG, "LE Advertise Failed: " + error);
278             }
279         }
280     };
```

```
281     /*
282      * Callback to handle incoming requests to the GATT server.
283      * All read/write requests for characteristics and descriptors are handled here.
284      */
285     private BluetoothGattServerCallback mGattServerCallback =
286     new BluetoothGattServerCallback() {
287
288         @Override
289         public void onConnectionStateChange(BluetoothDevice device,
290                                             int status, int newState) {
291             if (newState == BluetoothProfile.STATE_CONNECTED) {
292                 Log.i(TAG, "BluetoothDevice CONNECTED: " + device);
293                 mRegisteredDevices.add(device);
294             } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
295                 Log.i(TAG, "BluetoothDevice DISCONNECTED: " + device);
296                 //Remove device from any active subscriptions
297                 mRegisteredDevices.remove(device);
298             }
299         }
300
301         @Override
302         public void onCharacteristicReadRequest(
303                                             BluetoothDevice device,
304                                             int requestId, int offset,
305                                             BluetoothGattCharacteristic characteristic) {
306             if (RemoteSensorProfile
307                 .TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID
308                 .equals(characteristic.getUuid())) {
309                     Log.i(TAG, "Read characteristic data");
310                     mBluetoothGattServer.sendResponse(device,
311                         requestId,
312                         BluetoothGatt.GATT_SUCCESS,
313                         0,
314                         null);
315             } else {
316                 // Invalid characteristic
317                 Log.w(TAG, "Invalid characteristic read: " + characteristic.getUuid());
318                 mBluetoothGattServer.sendResponse(device,
319                         requestId,
320                         BluetoothGatt.GATT_FAILURE,
321                         0,
322                         null);
323             }
324         }
325
326
327         @Override
328         public void onDescriptorReadRequest(BluetoothDevice device,
329                                             int requestId, int offset,
330                                             BluetoothGattDescriptor descriptor) {
331             mBluetoothGattServer.sendResponse(device,
332                     requestId,
333                     BluetoothGatt.GATT_FAILURE,
334                     0,
335                     null);
336         }
337
338
```

```
339
340         @Override
341         public void onDescriptorWriteRequest(BluetoothDevice device, int requestId,
342                                              BluetoothGattDescriptor descriptor,
343                                              boolean preparedWrite,
344                                              boolean responseNeeded,
345                                              int offset, byte[] value) {
346
347             if (responseNeeded) {
348                 mBluetoothGattServer.sendResponse(device,
349                         requestId,
350                         BluetoothGatt.GATT_FAILURE,
351                         0,
352                         null);
353             }
354         }
355     };
356
357
358     private BluetoothPairingCallback mBluetoothPairingCallback =
359     new BluetoothPairingCallback() {
360
361         @Override
362         public void onPairingInitiated(BluetoothDevice bluetoothDevice,
363                                        PairingParams pairingParams) {
364             // Handle incoming pairing request or confirmation of
365             // outgoing pairing request
366             handlePairingRequest(bluetoothDevice, pairingParams);
367         }
368
369         @Override
370         public void onPaired(BluetoothDevice bluetoothDevice) {
371             Log.i(TAG,"Device pairing complete");
372         }
373
374         @Override
375         public void onUnpaired(BluetoothDevice bluetoothDevice) {
376             Log.i(TAG, "Device unpaired");
377         }
378
379         @Override
380         public void onPairingError(BluetoothDevice bluetoothDevice,
381                                    BluetoothPairingCallback.PairingError pairingError) {
382             Log.e(TAG,"Pairing Error "+pairingError.getErrorCode());
383         }
384     };
385
386     private void handlePairingRequest(BluetoothDevice bluetoothDevice,
387                                       PairingParams pairingParams) {
388         String pin;
389         switch (pairingParams.getPairingType()) {
390
391           case PairingParams.PAIRING_VARIANT_DISPLAY_PIN:
392           case PairingParams.PAIRING_VARIANT_DISPLAY_PASSKEY:
393                 // Display the required PIN to the user
394                 pin = pairingParams.getPairingPin();
395                 Log.d(TAG, "Display Passkey - " + pin);
396                 break;
397
```

```
398
399             case PairingParams.PAIRING_VARIANT_PIN:
400             case PairingParams.PAIRING_VARIANT_PIN_16_DIGITS:
401                     // Obtain PIN from the user
402                     pin = "0000";
403                     // Pass the result to complete pairing
404                     mBluetoothConnectionManager.finishPairing(bluetoothDevice, pin);
405                     break;
406             case PairingParams.PAIRING_VARIANT_CONSENT:
407                     mBluetoothConnectionManager.finishPairing(bluetoothDevice);
408                     break;
409             case PairingParams.PAIRING_VARIANT_PASSKEY_CONFIRMATION:
410                     // Show confirmation of pairing to the user:
411                     // Pairing Passkey is null and in the PicoPi
412                     // there is noconfirmation request.
413                     // Complete the pairing process
414                     mBluetoothConnectionManager.finishPairing(bluetoothDevice);
415                     break;
416         }
417     }
418
419     /*
420      * Send a TEMPERATURE_NOTIFY_CHARACTERISTIC to any devices
421      * that are subscribed to the characteristic.
422      */
423     private void notifyRegisteredDevices() {
424
425       if (mRegisteredDevices.isEmpty()) {
426               Log.i(TAG, "No subscribers registered");
427               return;
428       }
429
430       pairedDevices = mBluetoothAdapter.getBondedDevices();
431
432       // For each BT device registered
433       for (BluetoothDevice device : mRegisteredDevices) {
434
435         if(!pairedDevices.contains(device)){
436                 continue;
437         }
438
439         BluetoothGattCharacteristic temperatureDataCharacteristic =
440         mBluetoothGattServer
441          .getService(RemoteSensorProfile.REMOTE_SENSOR_SERVICE_UUID)
442          .getCharacteristic(RemoteSensorProfile
443                  .TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID);
444
445           float temp = readTemperature();
446           String temperature = "";
447           temperature = String.valueOf(temp);
448           temperatureDataCharacteristic.setValue(temperature);
449           boolean success = mBluetoothGattServer
450             .notifyCharacteristicChanged(device,temperatureDataCharacteristic,false);
451           Log.e(TAG, "temperature = "+temperature+" - SUCCESS = "+success);
452       }
453     }
454
455
456
```

```
457
458     /*
459      * Enable the current {@link BluetoothAdapter} to be discovered (available for
460      * pairing) for the next {@link #DISCOVERABLE_TIMEOUT_MS} ms.
461      */
462     private void enableDiscoverable() {
463         Log.d(TAG, "Registering for discovery.");
464
465         Intent discoverableIntent =
466         new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
467
468         discoverableIntent.putExtra(
469             BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,
470             DISCOVERABLE_TIMEOUT_MS);
471
472         startActivityForResult(discoverableIntent,
473                                REQUEST_CODE_ENABLE_DISCOVERABLE);
474     }
475
476
477     @Override
478     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
479
480         super.onActivityResult(requestCode, resultCode, data);
481
482         if (requestCode == REQUEST_CODE_ENABLE_DISCOVERABLE) {
483           Log.d(TAG, "Enable discoverable returned with result " + resultCode);
484
485           if (resultCode == RESULT_CANCELED) {
486               Log.e(TAG, "Enable discoverable has been cancelled by the user. " +
487                       "This should never happen in an Android Things device.");
488               return;
489           }
490
491           Log.i(TAG, "Bluetooth adapter successfully set to discoverable mode.\n " +
492                       "Any source can find it with the name " +
493                       ADAPTER_FRIENDLY_NAME +
494                       "\n and pair for the next " +
495                       DISCOVERABLE_TIMEOUT_MS + " ms. \n" +
496                       "Try looking for it on your phone, for example.");
497         }
498     }
499
500
501     private Bmx280 openTempSensor() throws IOException{
502         tempSensor = RainbowHat.openSensor();
503         tempSensor.setTemperatureOversampling(Bmx280.OVERSAMPLING_1X);
504         tempSensor.setMode(Bmx280.MODE_NORMAL);
505
506         return tempSensor;
507     }
508
509
510     private void closeTempSensor() throws IOException{
511         tempSensor.close();
512     }
513
514
515
```

```
516
517     private float readTemperature(){
518
519         float temperature = -274;
520
521         if (tempSensor != null){
522             try {
523                 temperature = tempSensor.readTemperature();
524                 BigDecimal tempBG = new BigDecimal(temperature);
525                 tempBG = tempBG.setScale(2, BigDecimal.ROUND_HALF_UP);
526                 temperature = (tempBG.floatValue());
527             }
528             catch (IOException e){
529                 Log.e(TAG, "Unable to read temperature from sensor");
530             }
531         }
532
533         return temperature;
534     }
535
536     private final Runnable reportTemperature = new Runnable() {
537         @Override
538         public void run() {
539           notifyRegisteredDevices();
540           handler.postDelayed(reportTemperature, TimeUnit.SECONDS.toMillis(2));
541         }
542     };
543 }
```

Listing A.7: MainActivity.java

RemoteSensorProfile

```
1  public class RemoteSensorProfile {
2
3      private final static String TAG = RemoteSensorProfile.class.getSimpleName();
4
5      // Remote Sensor Service UUID
6      public static UUID REMOTE_SENSOR_SERVICE_UUID =
7      UUID.fromString ("B340B65C-B8AE-49E7-8ED8-F79C61708475");
8
9      // Remote Sensor Data Characteristic UUID
10     public static UUID TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID =
11     UUID.fromString("FEE891B9-032A-43AF-8923-5E3A4FF989A3");
12
13     public static BluetoothGattService createRemoteSensorService() {
14
15       Log.d(TAG, "createRemoteSensorService()");
16
17       BluetoothGattService service = new BluetoothGattService(
18             REMOTE_SENSOR_SERVICE_UUID,
19             BluetoothGattService.SERVICE_TYPE_PRIMARY);
20
21       BluetoothGattCharacteristic characteristic = new BluetoothGattCharacteristic(
22             TEMPERATURE_NOTIFY_CHARACTERISTIC_UUID,
23             //Read-only characteristic, supports notifications
24             BluetoothGattCharacteristic.PROPERTY_READ |
25             BluetoothGattCharacteristic.PROPERTY_NOTIFY,
26             BluetoothGattCharacteristic.PERMISSION_READ_ENCRYPTED);
27
28       //characteristic.addDescriptor(config);
29       service.addCharacteristic(characteristic);
30
31       return service;
32     }
33 }
```

Listing A.8: RemoteSensorProfile.java

A.4.2 BLEclient (Android app)

DeviceScanActivity

```
public class DeviceScanActivity extends ListActivity {

    private static final String TAG = DeviceScanActivity.class.getSimpleName();

    private final int REQUEST_PERMISSION_POSITION=1;

    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private BluetoothLeScanner mBluetoothLeScanner;
    private LeDeviceListAdapter mLeDeviceListAdapter;

    private boolean mScanning = false;
    private Handler mHandler;

    private static final int REQUEST_ENABLE_BT = 1;

    // Stops scanning after 10 seconds.
    private static final long SCAN_PERIOD = 10000;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Use this check to determine whether BLE is supported on the device.
        // Then you can selectively disable BLE-related features.
        if (!getPackageManager()
            .hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
                Toast.makeText(this,
                                R.string.ble_not_supported,
                                Toast.LENGTH_SHORT).show();
                finish();
        }

        ActivityCompat.requestPermissions(DeviceScanActivity.this,
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                REQUEST_PERMISSION_POSITION);

        // Initializes a Bluetooth Manager.
        mBluetoothManager =
          (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);

        // Initializes a Bluetooth Adapter.
        mBluetoothAdapter = mBluetoothManager.getAdapter();

        // Initializes BluetoothLeScanner
        mBluetoothLeScanner = mBluetoothAdapter.getBluetoothLeScanner();

        mHandler = new Handler();

        registerReceiver(mAdapterStateChangeReceiver, new IntentFilter(
                BluetoothAdapter.ACTION_STATE_CHANGED));
    }


```

```
56      @Override
57      public boolean onCreateOptionsMenu(Menu menu) {
58          getMenuInflater().inflate(R.menu.main, menu);
59          if (!mScanning) {
60              menu.findItem(R.id.menu_stop).setVisible(false);
61              menu.findItem(R.id.menu_scan).setVisible(true);
62              menu.findItem(R.id.menu_refresh).setActionView(null);
63          } else {
64              menu.findItem(R.id.menu_stop).setVisible(true);
65              menu.findItem(R.id.menu_scan).setVisible(false);
66              menu.findItem(R.id.menu_refresh).setActionView(
67                      R.layout.actionbar_indeterminate_progress);
68          }
69          return true;
70      }
71
72      @Override
73      public boolean onOptionsItemSelected(MenuItem item) {
74          switch (item.getItemId()) {
75              case R.id.menu_scan:
76                  mLeDeviceListAdapter.clear();
77                  scanLeDevice(true);
78                  break;
79              case R.id.menu_stop:
80                  scanLeDevice(false);
81                  break;
82          }
83          return true;
84      }
85
86      @Override
87      protected void onResume() {
88          super.onResume();
89          // Ensures Bluetooth is enabled on the device. If Bluetooth is not currently
90          // enabled, fire an intent to display a dialog asking the user to grant
91          // permission to enable it.
92          if (!mBluetoothAdapter.isEnabled()) {
93              Intent enableBtIntent =
94              new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
95              startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
96          }
97          // Initializes list view adapter.
98          mLeDeviceListAdapter = new LeDeviceListAdapter();
99          setListAdapter(mLeDeviceListAdapter);
100         scanLeDevice(true);
101     }
102
103     @Override
104     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
105         // User chose not to enable Bluetooth.
106         if (requestCode == REQUEST_ENABLE_BT &&
107            resultCode == Activity.RESULT_CANCELED) {
108             finish();
109             return;
110         }
111         super.onActivityResult(requestCode, resultCode, data);
112     }
```

```
113
114     @Override
115     protected void onPause() {
116         super.onPause();
117         scanLeDevice(false);
118         mLeDeviceListAdapter.clear();
119     }
120
121     @Override
122     protected void onListItemClick(ListView l, View v, int position, long id) {
123       final BluetoothDevice device = mLeDeviceListAdapter.getDevice(position);
124       if (device == null) return;
125       final Intent intent = new Intent(this, MainActivity.class);
126       intent.putExtra(MainActivity.EXTRAS_DEVICE_OBJECT, device);
127       intent.putExtra(MainActivity.EXTRAS_DEVICE_NAME, device.getName());
128       intent.putExtra(MainActivity.EXTRAS_DEVICE_ADDRESS,
129                       device.getAddress());
130       if (mScanning) {
131             mBluetoothLeScanner.stopScan(mLeScanCallback);
132             mScanning = false;
133       }
134       startActivity(intent);
135     }
136
137     private void scanLeDevice(final boolean enable) {
138         if (enable) {
139             Log.d(TAG, "scanLeDevice - START");
140             // Stops scanning after a pre-defined scan period.
141             mHandler.postDelayed(new Runnable() {
142                 @Override
143                 public void run() {
144                     mScanning = false;
145                     mBluetoothLeScanner.stopScan(mLeScanCallback);
146                     invalidateOptionsMenu();
147                 }
148             }, SCAN_PERIOD);
149
150             mScanning = true;
151             mBluetoothLeScanner.startScan(mLeScanCallback);
152         } else {
153             Log.d(TAG, "scanLeDevice - STOP");
154             mScanning = false;
155             mBluetoothLeScanner.stopScan(mLeScanCallback);
156         }
157         invalidateOptionsMenu();
158     }
159
160     // Adapter for holding devices found through scanning.
161     private class LeDeviceListAdapter extends BaseAdapter {
162         private ArrayList<BluetoothDevice> mLeDevices;
163         private LayoutInflater mInflator;
164
165         public LeDeviceListAdapter() {
166             super();
167             mLeDevices = new ArrayList<BluetoothDevice>();
168             mInflator = DeviceScanActivity.this.getLayoutInflater();
169         }
```

```
170
171         public void addDevice(BluetoothDevice device) {
172             if(!mLeDevices.contains(device)) {
173                 mLeDevices.add(device);
174             }
175         }
176
177         public BluetoothDevice getDevice(int position) {
178             return mLeDevices.get(position);
179         }
180
181         public void clear() {
182             mLeDevices.clear();
183         }
184
185         @Override
186         public int getCount() {
187             return mLeDevices.size();
188         }
189
190         @Override
191         public Object getItem(int i) {
192             return mLeDevices.get(i);
193         }
194
195         @Override
196         public long getItemId(int i) {
197             return i;
198         }
199
200         @Override
201         public View getView(int i, View view, ViewGroup viewGroup) {
202             ViewHolder viewHolder;
203             // General ListView optimization code.
204             if (view == null) {
205                 view = mInflator.inflate(R.layout.activity_device_scan, null);
206                 viewHolder = new ViewHolder();
207                 viewHolder.deviceAddress =
208                         (TextView) view.findViewById(R.id.device_address);
209                 viewHolder.deviceName =
210                         (TextView) view.findViewById(R.id.device_name);
211                 view.setTag(viewHolder);
212             } else {
213                 viewHolder = (ViewHolder) view.getTag();
214             }
215
216             BluetoothDevice device = mLeDevices.get(i);
217             final String deviceName = device.getName();
218             if (deviceName != null && deviceName.length() > 0)
219                 viewHolder.deviceName.setText(deviceName);
220             else
221                 viewHolder.deviceName.setText(R.string.unknown_device);
222             viewHolder.deviceAddress.setText(device.getAddress());
223
224             return view;
225         }
226     }
227
```

```
228
229     // Device scan callback.
230     private ScanCallback mLeScanCallback = new ScanCallback() {
231         @Override
232         public void onScanResult(int callbackType, ScanResult result) {
233             super.onScanResult(callbackType, result);
234             Log.d(TAG,"Device found: "+result.getDevice().getName());
235             mLeDeviceListAdapter.addDevice(result.getDevice());
236             mLeDeviceListAdapter.notifyDataSetChanged();
237         }
238
239         @Override
240         public void onScanFailed(int errorCode) {
241             Log.e(TAG, "BLE Scan Failed with code " + errorCode);
242         }
243     };
244
245     static class ViewHolder {
246         TextView deviceName;
247         TextView deviceAddress;
248     }
249
250
251     /*
252      * Handle an intent that is broadcast by the Bluetooth adapter whenever it
253      * changes its state (after calling enable(), for example).
254      * Action is {@link BluetoothAdapter#ACTION_STATE_CHANGED} and
255      * extras describe the old and the new states. You can use this intent to
256      * indicate that the device is ready to go.
257      */
258     private final BroadcastReceiver mAdapterStateChangeReceiver =
259     new BroadcastReceiver() {
260         public void onReceive(Context context, Intent intent) {
261             int oldState = getPreviousAdapterState(intent);
262             int newState = getCurrentAdapterState(intent);
263             Log.d(TAG, "Bluetooth Adapter changing state from " +
264                         oldState + " to " + newState);
265             if (newState == BluetoothAdapter.STATE_ON) {
266                 Log.i(TAG, "Bluetooth Adapter is ready");
267             }
268         }
269     };
270
271   private int getPreviousAdapterState(Intent intent) {
272     return intent.getIntExtra(BluetoothAdapter.EXTRA_PREVIOUS_STATE, -1);
273   }
274
275   private int getCurrentAdapterState(Intent intent) {
276     return intent.getIntExtra(BluetoothAdapter.EXTRA_STATE, -1);
277   }
278 }
```

Listing A.9: DeviceScanActivity.java

MainActivity

```
1  public class MainActivity extends AppCompatActivity {
2
3    private static final String TAG = MainActivity.class.getSimpleName();
4
5    /* Remote Sensor Service UUID */
6    public static UUID REMOTE_SENSOR_SERVICE_UUID =
7    UUID.fromString ("B340B65C-B8AE-49E7-8ED8-F79C61708475");
8    /* Remote Sensor Data Characteristic UUID */
9    public static UUID TEMP_NOTIFY_CHARACTERISTIC_UUID =
10   UUID.fromString("FEE891B9-032A-43AF-8923-5E3A4FF989A3");
11
12   public static final String EXTRAS_DEVICE_OBJECT = "DEVICE_OBJECT";
13   public static final String EXTRAS_DEVICE_NAME = "DEVICE_NAME";
14   public static final String EXTRAS_DEVICE_ADDRESS ="DEVICE_ADDRESS";
15
16   private BluetoothGattCharacteristic mNotifyCharacteristic;
17   private String mRemoteDeviceName;
18   private String mRemoteDeviceAddress;
19   private BluetoothDevice mRemoteServer;
20   private BluetoothManager mBluetoothManager;
21   private BluetoothAdapter mBluetoothAdapter;
22   private BluetoothLeService mBluetoothLeService;
23
24   private boolean mConnected = false;
25   EditText tempValueEditText;
26
27     @Override
28     protected void onCreate(Bundle savedInstanceState) {
29
30         super.onCreate(savedInstanceState);
31         setContentView(R.layout.activity_main);
32
33         ActivityCompat.requestPermissions(
34             this,new String[]{permission.BLUETOOTH},1);
35         ActivityCompat.requestPermissions(
36             this,new String[]{permission.BLUETOOTH_ADMIN},1);
37         ActivityCompat.requestPermissions(
38             this,new String[]{permission.ACCESS_COARSE_LOCATION},1);
39
40
41         // Initializes a Bluetooth adapter.
42         mBluetoothManager =
43         (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
44         mBluetoothAdapter = mBluetoothManager.getAdapter();
45
46         tempValueEditText = (EditText) findViewById(R.id.TempEditText);
47
48         final Intent intent = getIntent();
49
50         mRemoteServer = getIntent()
51                         .getExtras().getParcelable("DEVICE_OBJECT");
52         mRemoteDeviceName = intent
53                         .getStringExtra(EXTRAS_DEVICE_NAME);
54         mRemoteDeviceAddress = intent.
55                         .getStringExtra(EXTRAS_DEVICE_ADDRESS);
56
57
```

```
58
59          // register pairing state receiver
60          IntentFilter bondChangedIntent =
61          new IntentFilter(BluetoothDevice.ACTION_BOND_STATE_CHANGED);
62          registerReceiver(bondChangedReceived, bondChangedIntent);
63
64          if (mBluetoothLeService != null) {
65            final boolean result=mBluetoothLeService.connect(mRemoteDeviceAddress);
66            Log.d(TAG, "Connect request result=" + result);
67          }
68
69          if(mRemoteServer == null){
70              Log.e(TAG, "Unable to find BLE server!");
71              MainActivity.this.finish();
72          }
73
74          // Start the connection (automatic pairing starts if necessary)
75          Intent gattServiceIntent =
76          new Intent(MainActivity.this, BluetoothLeService.class);
77          bindService(gattServiceIntent,mServiceConnection,BIND_AUTO_CREATE);
78
79          /*
80          // Checks if devices are bondend.
81          // Initiates a pairing phase or starts the connection
82          Set<BluetoothDevice> pairedDevices =
83              mBluetoothAdapter.getBondedDevices();
84          if(pairedDevices.size() > 0 && pairedDevices.contains(mRemoteServer)) {
85              // start the connection
86              if (!mConnected) {
87                  Intent gattServiceIntent =
88                  new Intent(MainActivity.this, BluetoothLeService.class);
89                  bindService(gattServiceIntent,
90                              mServiceConnection,
91                              BIND_AUTO_CREATE);
92              }
93          }
94          else {
95              // start pairing
96              mRemoteServer.createBond();
97          }
98          */
99      }
100
101     @Override
102     protected void onStart() {
103         super.onStart();
104         Log.d(TAG, "onStart()");
105         registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilter());
106     }
107
108     @Override
109     protected void onStop(){
110         super.onStop();
111         Log.d(TAG, "onPause()");
112         unregisterReceiver(mGattUpdateReceiver);
113     }
114
```

```
115
116     @Override
117     protected void onDestroy() {
118         super.onDestroy();
119         Log.d(TAG, "onDestroy()");
120         if(mServiceConnection != null)
121             unbindService(mServiceConnection);
122         unregisterReceiver(bondChangedReceived);
123     }
124
125
126     private BroadcastReceiver bondChangedReceived = new BroadcastReceiver() {
127
128         @Override
129         public void onReceive(Context context, Intent intent) {
130           final String action = intent.getAction();
131           Log.d(TAG, "onReceive() action: "+action);
132
133           if(action.equals(BluetoothDevice.ACTION_BOND_STATE_CHANGED)){
134                 Log.d(TAG, "Changed bond state");
135                 BluetoothDevice device =
136                 intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
137
138                 if (device.getBondState() == BluetoothDevice.BOND_BONDING) {
139                     Log.d(TAG, " BOND_BONDING with "+device.getName());
140                 }
141
142                 if (device.getBondState() == BluetoothDevice.BOND_NONE) {
143                     Log.d(TAG, " BOND_NONE");
144                 }
145
146                 if (device.getBondState() == BluetoothDevice.ERROR) {
147                     Log.d(TAG, " ERROR");
148                 }
149
150                 if (device.getBondState() == BluetoothDevice.BOND_BONDED) {
151                     Log.d(TAG, " BOND with "+device.getName());
152                 }
153           }
154         }
155     };
156
157
158     private static IntentFilter makeGattUpdateIntentFilter() {
159         final IntentFilter intentFilter = new IntentFilter();
160         intentFilter.addAction(BluetoothLeService
161                                .ACTION_GATT_CONNECTED);
162         intentFilter.addAction(BluetoothLeService
163                                .ACTION_GATT_DISCONNECTED);
164         intentFilter.addAction(BluetoothLeService
165                                .ACTION_GATT_SERVICES_DISCOVERED);
166         intentFilter.addAction(BluetoothLeService
167                                .ACTION_DATA_AVAILABLE);
168
169         return intentFilter;
170     }
171
172
```

```
173
174   // Handles various events fired by the Service.
175   private BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
176     @Override
177     public void onReceive(Context context, Intent intent) {
178       final String action = intent.getAction();
179       Log.d(TAG, "onReceive() action: "+action);
180
181       if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
182           mConnected = true;
183       }
184       else if(BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action)){
185           mConnected = false;
186       }
187
188       else if (BluetoothLeService
189               .ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
190
191           // Show all the supported services and charac. on the user interface.
192           List<BluetoothGattService> services =
193           mBluetoothLeService.getSupportedGattServices();
194           if(services != null){
195               for (BluetoothGattService gattService : services) {
196                   if(gattService.getUuid()
197                     .equals(REMOTE_SENSOR_SERVICE_UUID)){
198
199                     final BluetoothGattCharacteristic characteristic =
200                     gattService.getCharacteristic(
201                     TEMP_NOTIFY_CHARACTERISTIC_UUID);
202
203                      if (characteristic != null) {
204                        final int charaProp = characteristic.getProperties();
205                        if ((charaProp |
206                            BluetoothGattCharacteristic.PROPERTY_READ) > 0) {
207                            // If there is an active notification on a characteristic,
208                            // clear it first so it doesn't update the data field on
209                            // the user interface.
210                            if (mNotifyCharacteristic != null) {
211                                mBluetoothLeService.setCharacteristicNotification(
212                                    mNotifyCharacteristic, false);
213
214                                mNotifyCharacteristic = null;
215                            }
216
217                            mBluetoothLeService.readCharacteristic(characteristic);
218                        }
219                        if ((charaProp |
220                            BluetoothGattCharacteristic.PROPERTY_NOTIFY)>0){
221
222                            mNotifyCharacteristic = characteristic;
223                            mBluetoothLeService
224                              .setCharacteristicNotification(characteristic, true);
225                        }
226                      }
227                   }
228               }
229           }
230       }
```

```
231
232         else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
233             String data = intent.getStringExtra(BluetoothLeService.EXTRA_DATA);
234             if (data == null)
235               return;
236             if(Float.parseFloat(data) <= -274.0f)
237               tempValueEditText.setText("error", TextView.BufferType.EDITABLE);
238             else {
239               tempValueEditText.setText(data, TextView.BufferType.EDITABLE);
240               Log.d(TAG, "temperature = "+data);
241             }
242         }
243       }
244     };
245
246     private final ServiceConnection mServiceConnection = new ServiceConnection() {
247
248       @Override
249       public void onServiceConnected(ComponentName componentName,
250                                      IBinder service) {
251         Log.d(TAG, "onServiceConnected()");
252         mBluetoothLeService = ((BluetoothLeService.LocalBinder) service).getService();
253         if (!mBluetoothLeService.initialize()) {
254             Log.e(TAG, "Unable to initialize Bluetooth");
255             finish();
256         }
257
258         Boolean result = mBluetoothLeService.connect(mRemoteDeviceAddress);
259         Log.d(TAG, "Connect request result=" + result);
260         mConnected = true;
261       }
262
263       @Override
264       public void onServiceDisconnected(ComponentName componentName) {
265         Log.d(TAG, "onServiceDisconnected()");
266         mBluetoothLeService = null;
267         mConnected = false;
268       }
269     };
270 }
```

Listing A.10: MainActivity.java

A.5 BLEPicoClient

A.5.1 PicoPiClient

MainActivity

```
1  public class MainActivity extends Activity {
2
3      private static final String TAG = MainActivity.class.getSimpleName();
4
5      private Gpio ledG;
6      private Gpio ledR;
7      private Gpio ledB;
8
9      // Remote LED Service UUID
10     public static UUID REMOTE_LED_SERVICE =
11     UUID.fromString("00001805-0000-1000-8000-00805f9b34fb");
12     // Remote LED Data Characteristic
13     public static UUID REMOTE_LED_DATA =
14     UUID.fromString("00002a2b-0000-1000-8000-00805f9b34fb");
15
16     private static final String ADAPTER_FRIENDLY_NAME = "PicoPiDevice";
17     private static final String ANDROID_DEVICE_NAME = "S8AlessandroBLE";
18     private static final int DISCOVERABLE_TIMEOUT_MS = 60000;
19     private static final long SCAN_PERIOD = 10000;
20     private static final int REQUEST_CODE_ENABLE_DISCOVERABLE = 100;
21
22     private BluetoothAdapter mBluetoothAdapter;
23     private BluetoothProfileManager mBluetoothProfileManager;
24     private BluetoothConnectionManager mBluetoothConnectionManager;
25     private BluetoothConfigManager mBluetoothConfigManager;
26     private BluetoothLeService mBluetoothLeService;
27
28     // RainbowHat Button
29     private Button mScanningButton;
30     private Button mCloseButton;
31
32     // Android Button on lcd
33     android.widget.Button lcdButton;
34     private TextView mPasswordTextView;
35
36     // Set which contains paired devices
37     private Set<BluetoothDevice> pairedDevices;
38
39     private boolean mConnected = false;
40     private boolean mScanning = false;
41
42     private String mRemoteDeviceAddress;
43     private BluetoothDevice mRemoteBluetoothDevice;
44
45     private BluetoothGattCharacteristic mNotifyCharacteristic;
46
47     private Handler mHandler;
48     private Handler pairingHandler;
49     public static HandlerThread pairingThread;
```

```
50
51      /**
52       * Handle an intent that is broadcast by the Bluetooth adapter whenever it
53       * changes its state (after calling enable(), for example).
54       * Action is {@link BluetoothAdapter#ACTION_STATE_CHANGED} and
55       * extras describe the old and the new states. You can use this intent to
56       * indicate that the device is ready to go.
57       */
58      private final BroadcastReceiver mAdapterStateChangeReceiver =
59      new BroadcastReceiver() {
60          public void onReceive(Context context, Intent intent) {
61              int oldState = getPreviousAdapterState(intent);
62              int newState = getCurrentAdapterState(intent);
63              Log.d(TAG, "Bluetooth Adapter changing state from " +
64                              oldState + " to " + newState);
65              if (newState == BluetoothAdapter.STATE_ON) {
66                  Log.i(TAG, "Bluetooth Adapter is ready");
67                  init();
68              }
69          }
70      };
71
72      private BroadcastReceiver bluetoothDeviceReceiver = new BroadcastReceiver() {
73          @Override
74          public void onReceive(Context context, Intent intent) {
75              if (BluetoothDevice.ACTION_BOND_STATE_CHANGED
76              .equals( intent.getAction() ) ) {
77                  BluetoothDevice device =
78                  intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
79                  switch (device.getBondState()) {
80                      case BluetoothDevice.BOND_BONDING:
81                          Log.d(TAG, "Start pairing with "+device.getName()+": "+
82                                          device.getAddress());
83                          break;
84                      case BluetoothDevice.BOND_BONDED:
85                          Log.d(TAG, "Start pairing with "+device.getName()+": "+
86                                          device.getAddress());
87                          break;
88                      case BluetoothDevice.BOND_NONE:
89                          Log.d(TAG, "cancel");
90                          break;
91                      case BluetoothDevice.ERROR:
92                          Log.d(TAG, "error");
93                      default:
94                          break;
95                  }
96
97              }
98          }
99      };
```

```
100     private BluetoothPairingCallback mBluetoothPairingCallback =
101     new BluetoothPairingCallback() {
102
103         @Override
104         public void onPairingInitiated(BluetoothDevice bluetoothDevice,
105                                        PairingParams pairingParams) {
106             Log.i(TAG, "Handle incoming pairing request or confirmation of
107                         outgoing pairing request");
108
109             // Create a new thread for pairing management
110             pairingThread = new HandlerThread("pairingThread");
111             // Start the thread
112             pairingThread.start();
113             // Create an Handler for pairing management
114             pairingHandler = new Handler(pairingThread.getLooper());
115             // Start the handler
116             pairingHandler.post(new HandlePairingRequestRunnable(bluetoothDevice,
117                                                                  pairingParams));
118         }
119
120         @Override
121         public void onPaired(BluetoothDevice bluetoothDevice) {
122             // Device pairing complete
123             Log.i(TAG,"Device pairing complete");
124         }
125
126         @Override
127         public void onUnpaired(BluetoothDevice bluetoothDevice) {
128             // Device unpaired
129             Log.i(TAG, "Device unpaired");
130         }
131
132         @Override
133         public void onPairingError(BluetoothDevice bluetoothDevice,
134                                    BluetoothPairingCallback.PairingError pairingError) {
135             // Something went wrong!
136             Log.e(TAG,"onPairingError "+pairingError.getErrorCode());
137         }
138     };
139
140
141     @Override
142     protected void onCreate(Bundle savedInstanceState) {
143         super.onCreate(savedInstanceState);
144         setContentView(R.layout.activity_main);
145
146         // Turn LEDs off
147         turnOffLeds();
148
149         mPasswordTextView = (TextView) findViewById(R.id.password);
150
151         mBluetoothProfileManager = BluetoothProfileManager.getInstance();
152         mBluetoothConnectionManager = BluetoothConnectionManager.getInstance();
153         mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
154         mBluetoothConfigManager = BluetoothConfigManager.getInstance();
155         mHandler = new Handler();
156
```

```
157
158         /* Set I/O capability of the device */
159
160         // With this capability it is necessary to use the LCD display
161         mBluetoothConfigManager
162             .setLeIoCapability(BluetoothConfigManager.IO_CAPABILITY_IN);
163
164         /* Other possibility for the IO capability
165          * mBluetoothConfigManager
166          * .setLeIoCapability(BluetoothConfigManager.IO_CAPABILITY_OUT); */
167
168         /* This capability (IO_CAPABILITY_NONE) return an error
169          * mBluetoothConfigManager
170          * .setLeIoCapability(BluetoothConfigManager.IO_CAPABILITY_NONE);*/
171
172
173         if (mBluetoothAdapter == null) {
174             Log.w(TAG, "No default Bluetooth adapter.
175                         Device likely does not support bluetooth.");
176             return;
177         }
178
179         registerReceiver(mAdapterStateChangeReceiver,
180                 new IntentFilter(BluetoothAdapter
181                                         .ACTION_STATE_CHANGED));
182
183         registerReceiver(bluetoothDeviceReceiver,
184                 new IntentFilter(BluetoothDevice
185                                         .ACTION_BOND_STATE_CHANGED));
186
187         mBluetoothConnectionManager
188             .registerPairingCallback(mBluetoothPairingCallback);
189
190
191         if (mBluetoothAdapter.isEnabled()) {
192             Log.d(TAG, "Bluetooth Adapter is already enabled.");
193             init();
194         }
195         else {
196             Log.d(TAG, "Bluetooth adapter not enabled. Enabling.");
197             mBluetoothAdapter.enable();
198         }
199     }
200
201     private static IntentFilter makeGattUpdateIntentFilter() {
202         final IntentFilter intentFilter = new IntentFilter();
203         intentFilter.addAction(BluetoothLeService
204                                 .ACTION_GATT_CONNECTED);
205         intentFilter.addAction(BluetoothLeService
206                                 .ACTION_GATT_DISCONNECTED);
207         intentFilter.addAction(BluetoothLeService
208                                 .ACTION_GATT_SERVICES_DISCOVERED);
209         intentFilter.addAction(BluetoothLeService
210                                 .ACTION_DATA_AVAILABLE);
211         return intentFilter;
212     }
213
214
```

```
215
216     @Override
217     protected void onStart() {
218         super.onStart();
219         Log.d(TAG, "onStart()");
220         registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilter());
221     }
222
223     @Override
224     protected void onPause(){
225         super.onPause();
226         Log.d(TAG, "onPause()");
227         unregisterReceiver(mGattUpdateReceiver);
228     }
229
230
231     @Override
232     protected void onDestroy() {
233         super.onDestroy();
234         Log.d(TAG, "onDestroy");
235
236         try {
237             mScanningButton.close();
238             mCloseButton.close();
239         }
240         catch (IOException e) {
241             Log.e(TAG, "Unable to close button");
242         }
243
244         if(mServiceConnection != null)
245             unbindService(mServiceConnection);
246
247         unregisterReceiver(mAdapterStateChangeReceiver);
248         unregisterReceiver(bluetoothDeviceReceiver);
249
250         mBluetoothConnectionManager
251             .unregisterPairingCallback(mBluetoothPairingCallback);
252
253         mBluetoothAdapter.disable();
254     }
255
256
257     private int getPreviousAdapterState(Intent intent) {
258       return intent.getIntExtra(BluetoothAdapter.EXTRA_PREVIOUS_STATE,-1);
259     }
260
261     private int getCurrentAdapterState(Intent intent) {
262       return intent.getIntExtra(BluetoothAdapter.EXTRA_STATE, -1);
263     }
```

```
264  public class HandlePairingRequestRunnable implements Runnable {
265
266      BluetoothDevice bluetoothDevice;
267      PairingParams pairingParams;
268
269      public HandlePairingRequestRunnable(BluetoothDevice bluetoothDevice,
270                                          PairingParams pairingParams) {
271          this.bluetoothDevice = bluetoothDevice;
272          this.pairingParams = pairingParams;
273      }
274      public void run() {
275        switch (pairingParams.getPairingType()) {
276         case PairingParams.PAIRING_VARIANT_DISPLAY_PIN:
277         case PairingParams.PAIRING_VARIANT_DISPLAY_PASSKEY:
278                 // Display the required PIN to the user
279                 Log.d(TAG, "Display Passkey - "+pairingParams.getPairingPin());
280                 break;
281         case PairingParams.PAIRING_VARIANT_PIN:
282         case PairingParams.PAIRING_VARIANT_PIN_16_DIGITS:
283                 // Obtain PIN from the user
284                 String pin = "";
285                 // This check is only for debug purpose:
286                 // if IO_CAPABILITY_NONE is set, this
287                 // 'case' should not be reached
288                 if(mBluetoothConfigManager.getLeIoCapability() ==
289                   BluetoothConfigManager.IO_CAPABILITY_NONE)
290                     pin = "0000";
291                 else {
292                     synchronized (lcdButton) {
293                         try {
294                             lcdButton.wait();
295                         } catch (InterruptedException e) {
296                             Log.e(TAG, Thread.currentThread().getName() +
297                                     " error: " + e.toString());
298                         }
299                     }
300                     pin = mPasswordTextView.getText().toString();
301                 }
302                 Log.d(TAG, "pin = "+pin);
303                 // Pass the result to complete pairing
304                 mBluetoothConnectionManager.finishPairing(bluetoothDevice, pin);
305                 break;
306
307         case PairingParams.PAIRING_VARIANT_CONSENT:
308                 // Complete the pairing process
309                 mBluetoothConnectionManager.finishPairing(bluetoothDevice);
310                 break;
311
312         case PairingParams.PAIRING_VARIANT_PASSKEY_CONFIRMATION:
313                 // Show confirmation of pairing to the user
314                 // Complete the pairing process
315                 mBluetoothConnectionManager.finishPairing(bluetoothDevice);
316                 break;
317          }
318      }
319  }
320
321
```

```
322
323     private void init() {
324         if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
325             Log.e(TAG, "Bluetooth adapter not available or not enabled.");
326             return;
327         }
328         Log.d(TAG, "Set up Bluetooth Adapter name ");
329         mBluetoothAdapter.setName(ADAPTER_FRIENDLY_NAME);
330         // configure button
331         configureButton();
332     }
333
334     private void configureButton() {
335         try {
336             mScanningButton = RainbowHat.openButtonA();
337             mScanningButton
338             .setOnButtonEventListener(new Button.OnButtonEventListener() {
339                 @Override
340                 public void onButtonEvent(Button button, boolean pressed) {
341                     Log.d(TAG, "button A pressed");
342
343                     if(!mScanning) {
344                         // Enable Pairing mode (discoverable)
345                         enableDiscoverable();
346                         scanLeDevice();
347                     }
348
349                     else
350                         Log.d(TAG,"Scan already started!");
351                 }
352             });
353
354             mCloseButton = RainbowHat.openButtonC();
355             mCloseButton
356             .setOnButtonEventListener(new Button.OnButtonEventListener() {
357                 @Override
358                 public void onButtonEvent(Button button, boolean pressed) {
359                     Log.d(TAG, "button C pressed");
360                     //Close the application
361                     MainActivity.this.finish();
362                 }
363             });
364
365             lcdButton = (android.widget.Button) findViewById(R.id.lcdButton);
366             lcdButton.setOnClickListener(new View.OnClickListener() {
367                 @Override
368                 public void onClick(View v) {
369                     synchronized (lcdButton) { lcdButton.notify(); }
370                 }
371             });
372         }
373         catch (IOException e){
374             Log.e(TAG, "Unable to open button");
375         }
376     }
377
378
```

```
379
380     private void scanLeDevice() {
381         Log.d(TAG, "scanLeDevice()");
382         // Stops scanning after a pre-defined scan period.
383         mHandler.postDelayed(new Runnable() {
384             @Override
385             public void run() {
386                 mScanning = false;
387                 mBluetoothAdapter.stopLeScan(mLeScanCallback);
388             }
389         }, SCAN_PERIOD);
390
391         mScanning = true;
392         mBluetoothAdapter.startLeScan(mLeScanCallback);
393     }
394
395     private BluetoothAdapter.LeScanCallback mLeScanCallback =
396             new BluetoothAdapter.LeScanCallback() {
397
398                 @Override
399                 public void onLeScan(final BluetoothDevice device,
400                                      int rssi, byte[] scanRecord) {
401                     Boolean isDeviceFound = false;
402
403                     if(device != null){
404                         final String deviceName = device.getName();
405                         if (deviceName != null && deviceName.length() > 0) {
406                             Log.d(TAG, "Found device: "+deviceName);
407                             if(deviceName.replaceAll("\\s+", "")
408                               .equals(ANDROID_DEVICE_NAME)){
409                                 isDeviceFound = true;
410                                 mRemoteBluetoothDevice = device;
411                                 mRemoteDeviceAddress = device.getAddress();
412                             }
413                         }
414                     }
415
416                     if(isDeviceFound && !mConnected) {
417                       Set<BluetoothDevice> pairedDevices =
418                       mBluetoothAdapter.getBondedDevices();
419                       boolean b = pairedDevices.contains(mRemoteBluetoothDevice);
420                       Log.e(TAG, "TEST lista dispositivi: "+b);
421                       if(b){
422                           Intent gattServiceIntent =
423                           new Intent(MainActivity.this, BluetoothLeService.class);
424                           bindService(gattServiceIntent,
425                                       mServiceConnection,
426                                       BIND_AUTO_CREATE);
427                       }
428                       else{
429                           mBluetoothConnectionManager
430                              .initiatePairing(mRemoteBluetoothDevice);
431                       }
432                     }
433                 }
434             };
435
436
```

```
437
438     private final ServiceConnection mServiceConnection = new ServiceConnection() {
439
440         @Override
441         public void onServiceConnected(ComponentName componentName,
442                                        IBinder service) {
443             Log.d(TAG, "onServiceConnected()");
444             mBluetoothLeService =
445             ((BluetoothLeService.LocalBinder) service).getService();
446             if (!mBluetoothLeService.initialize()) {
447                 Log.e(TAG, "Unable to initialize Bluetooth");
448                 finish();
449             }
450
451             Boolean result = mBluetoothLeService.connect(mRemoteDeviceAddress);
452             Log.d(TAG, "Connect request result=" + result);
453             mConnected = true;
454             if(mScanning) {
455                 mScanning = false;
456                 mBluetoothAdapter.stopLeScan(mLeScanCallback);
457             }
458         }
459
460         @Override
461         public void onServiceDisconnected(ComponentName componentName) {
462             Log.d(TAG, "onServiceDisconnected()");
463             mBluetoothLeService = null;
464         }
465     };
466
467     // Handles various events fired by the Service.
468     // ACTION_GATT_CONNECTED: connected to a GATT server.
469     // ACTION_GATT_DISCONNECTED: disconnected from a GATT server.
470     // ACTION_GATT_SERVICES_DISCOVERED: discovered GATT services.
471     // ACTION_DATA_AVAILABLE: received data from the device.
472     // This can be a result of read or notification operations.
473
474     private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver(){
475         @Override
476         public void onReceive(Context context, Intent intent) {
477             Log.d(TAG, "onReceive()");
478
479             final String action = intent.getAction();
480
481             if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
482                 mConnected = true;
483
484             }
485             else if (BluetoothLeService
486                     .ACTION_GATT_DISCONNECTED.equals(action)) {
487                 mConnected = false;
488             }
```

```
489         else if (BluetoothLeService
490                 .ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
491
492             // Show all the supported services and charac. on the user interface.
493             List<BluetoothGattService> services =
494             mBluetoothLeService.getSupportedGattServices();
495             if(services != null){
496                 for (BluetoothGattService gattService : services) {
497                     if(gattService.getUuid().equals(REMOTE_LED_SERVICE)){
498
499                       final BluetoothGattCharacteristic characteristic =
500                       gattService.getCharacteristic(
501                       REMOTE LED DATA);
502
503                        if (characteristic != null) {
504                          final int charaProp = characteristic.getProperties();
505                          if ((charaProp |
506                              BluetoothGattCharacteristic.PROPERTY_READ) > 0) {
507                              // If there is an active notification on a characteristic,
508                              // clear it first so it doesn't update the data field on
509                              // the user interface.
510                              if (mNotifyCharacteristic != null) {
511                                  mBluetoothLeService.setCharacteristicNotification(
512                                      mNotifyCharacteristic, false);
513
514                                  mNotifyCharacteristic = null;
515                              }
516
517                              mBluetoothLeService.readCharacteristic(characteristic);
518                          }
519                          if ((charaProp |
520                              BluetoothGattCharacteristic.PROPERTY_NOTIFY)>0){
521
522                              mNotifyCharacteristic = characteristic;
523                              mBluetoothLeService
524                                .setCharacteristicNotification(characteristic, true);
525                          }
526                      }
527                  }
528              }
529          }
530      }
531
532      else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
533          String data = intent.getStringExtra(BluetoothLeService.EXTRA_DATA);
534          if (data == null)
535              return;
536          if(data.contains("false")){
537              setLedValue(false);
538          }else if(data.contains("true")){
539              setLedValue(true);
540          }
541       }
542  };
543
544
545
```

```
546
547     private void setLedValue(boolean value) {
548         try {
549             ledG = RainbowHat.openLedGreen();
550             ledG.setValue(value);
551             ledG.close();
552         }
553         catch (IOException e){
554             Log.e(TAG, "Error updating value of led "+String.valueOf(ledG));
555         }
556     }
557     private void turnOffLeds() {
558         try {
559             ledG = RainbowHat.openLedGreen();
560             ledG.setValue(false);
561             ledG.close();
562             ledR = RainbowHat.openLedRed();
563             ledR.setValue(false);
564             ledR.close();
565             ledB = RainbowHat.openLedBlue();
566             ledB.setValue(false);
567             ledB.close();
568         } catch (IOException e) {
569             Log.e(TAG, "Unable to turn off leds'light: " + e);
570         }
571     }
572     // Enable the current {@link BluetoothAdapter} to be discovered for the next
573     // {@link #DISCOVERABLE_TIMEOUT_MS} ms.
574     private void enableDiscoverable() {
575         Log.d(TAG, "Registering for discovery.");
576         Intent discoverableIntent =
577         new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
578         discoverableIntent.putExtra(BluetoothAdapter
579             .EXTRA_DISCOVERABLE_DURATION,
580             DISCOVERABLE_TIMEOUT_MS);
581         startActivityForResult(discoverableIntent,
582                                REQUEST_CODE_ENABLE_DISCOVERABLE);
583     }
584     @Override
585     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
586         super.onActivityResult(requestCode, resultCode, data);
587         if (requestCode == REQUEST_CODE_ENABLE_DISCOVERABLE) {
588             Log.d(TAG, "Enable discoverable returned with result " + resultCode);
589             if (resultCode == RESULT_CANCELED) {
590                 Log.e(TAG, "Enable discoverable has been cancelled by the user. " +
591                         "This should never happen in an Android Things device.");
592                 return;
593             }
594             Log.i(TAG, "Bluetooth adapter successfully set to discoverable mode. ");
595         }
596     }
597 }
```

Listing A.11: MainActivity.java

BluetoothLeService

```
public class BluetoothLeService extends Service{

    private final static String TAG = BluetoothLeService.class.getSimpleName();

    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private String mBluetoothDeviceAddress;
    private BluetoothGatt mBluetoothGatt;
    private int mConnectionState = STATE_DISCONNECTED;

    private static final int STATE_DISCONNECTED = 0;
    private static final int STATE_CONNECTING = 1;
    private static final int STATE_CONNECTED = 2;

    public final static String ACTION_GATT_CONNECTED =
            "com.example.bluetooth.le.ACTION_GATT_CONNECTED";
    public final static String ACTION_GATT_DISCONNECTED =
            "com.example.bluetooth.le.ACTION_GATT_DISCONNECTED";
    public final static String ACTION_GATT_SERVICES_DISCOVERED =
            "com.example.bluetooth.le.ACTION_GATT_SERVICES_DISCOVERED";
    public final static String ACTION_DATA_AVAILABLE =
            "com.example.bluetooth.le.ACTION_DATA_AVAILABLE";
    public final static String EXTRA_DATA =
            "com.example.bluetooth.le.EXTRA_DATA";

    // Implements callback methods for GATT events that the app cares about.
    // For example, connection change and services discovered.
    private final BluetoothGattCallback mGattCallback =
    new BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt,
                                            int status, int newState) {
            String intentAction;
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                intentAction = ACTION_GATT_CONNECTED;
                mConnectionState = STATE_CONNECTED;
                broadcastUpdate(intentAction);
                Log.i(TAG, "Connected to GATT server.");
                // Attempts to discover services after successful connection.
                Log.i(TAG, "Attempting to start service discovery:" +
                        mBluetoothGatt.discoverServices());

            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                intentAction = ACTION_GATT_DISCONNECTED;
                mConnectionState = STATE_DISCONNECTED;
                Log.i(TAG, "Disconnected from GATT server.");
                broadcastUpdate(intentAction);
            }
        }
        @Override
        public void onServicesDiscovered(BluetoothGatt gatt, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
            } else {
                Log.w(TAG, "onServicesDiscovered received: " + status);
            }
        }
```

```
58
59          @Override
60          public void onCharacteristicRead(BluetoothGatt gatt,
61                                           BluetoothGattCharacteristic characteristic,
62                                           int status) {
63              if (status == BluetoothGatt.GATT_SUCCESS) {
64                  broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
65              }
66          }
67
68          @Override
69          public void onCharacteristicChanged(BluetoothGatt gatt,
70                                              BluetoothGattCharacteristic characteristic) {
71              broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
72          }
73      };
74
75      private void broadcastUpdate(final String action) {
76          final Intent intent = new Intent(action);
77          sendBroadcast(intent);
78      }
79
80      private void broadcastUpdate(final String action,
81                                   final BluetoothGattCharacteristic characteristic) {
82          final Intent intent = new Intent(action);
83          // For all other profiles, writes the data formatted in HEX.
84          final byte[] data = characteristic.getValue();
85          if (data != null && data.length > 0) {
86              final StringBuilder stringBuilder = new StringBuilder(data.length);
87              for(byte byteChar : data)
88                  stringBuilder.append(String.format("%02X ", byteChar));
89              intent.putExtra(EXTRA_DATA, new String(data) + "\n" +
90                                       stringBuilder.toString());
91          }
92          sendBroadcast(intent);
93      }
94
95      public class LocalBinder extends Binder {
96          BluetoothLeService getService() {
97              return BluetoothLeService.this;
98          }
99      }
100
101     @Override
102     public IBinder onBind(Intent intent) {
103         return mBinder;
104     }
105
106     @Override
107     public boolean onUnbind(Intent intent) {
108         // After using a given device, you should make sure that
109         // BluetoothGatt.close() is called such that resources are
110         // cleaned up properly. In this particular example, close()
111         // is invoked when the UI is disconnected from the Service.
112         close();
113         return super.onUnbind(intent);
114     }
```

```
115
116     private final IBinder mBinder = new LocalBinder();
117
118     /**
119      * Initializes a reference to the local Bluetooth adapter.
120      *
121      * @return Return true if the initialization is successful.
122      */
123     public boolean initialize() {
124         // For API level 18 and above, get a reference to BluetoothAdapter through
125         // BluetoothManager.
126         if (mBluetoothManager == null) {
127             mBluetoothManager =
128             (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
129             if (mBluetoothManager == null) {
130                 Log.e(TAG, "Unable to initialize BluetoothManager.");
131                 return false;
132             }
133         }
134
135         mBluetoothAdapter = mBluetoothManager.getAdapter();
136         if (mBluetoothAdapter == null) {
137             Log.e(TAG, "Unable to obtain a BluetoothAdapter.");
138             return false;
139         }
140
141         return true;
142     }
143
144     /**
145      * Connects to the GATT server hosted on the Bluetooth LE device.
146      *
147      * @param address The device address of the destination device.
148      *
149      * @return Return true if the connection is initiated successfully.
150      * The connection result is reported asynchronously through the
151      * {@code BluetoothGattCallback#onConnectionStateChange(
152      * android.bluetooth.BluetoothGatt, int, int)} callback.
153      */
154     public boolean connect(final String address) {
155         Log.d(TAG, "connect()");
156
157         if (mBluetoothAdapter == null || address == null) {
158             Log.w(TAG, "BluetoothAdapter not initialized or unspecified address.");
159             return false;
160         }
161         // Previously connected device. Try to reconnect.
162         if (mBluetoothDeviceAddress != null &&
163             address.equals(mBluetoothDeviceAddress)
164             && mBluetoothGatt != null) {
165             Log.d(TAG, "Trying to use an existing mBluetoothGatt for connection.");
166             if (mBluetoothGatt.connect()) {
167                 mConnectionState = STATE_CONNECTING;
168                 return true;
169             } else {
170                 return false;
171             }
172         }
173
```

```
174
175         final BluetoothDevice device = mBluetoothAdapter.getRemoteDevice(address);
176         if (device == null) {
177             Log.w(TAG, "Device not found. Unable to connect.");
178             return false;
179         }
180         // We want to directly connect to the device, so we are setting
181         // the autoConnect parameter to false.
182         // mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
183
184         mBluetoothGatt =
185         (new BleConnectionCompat(this)).connectGatt(device, false, mGattCallback);
186         Log.d(TAG, "Trying to create a new connection.");
187         mBluetoothDeviceAddress = address;
188         mConnectionState = STATE_CONNECTING;
189         return true;
190     }
191     /**
192      * Disconnects an existing connection or cancel a pending connection.
193      * The disconnection result is reported asynchronously through the
194      * {@code BluetoothGattCallback#onConnectionStateChange(
195      * android.bluetooth.BluetoothGatt, int, int)} callback.
196      */
197     public void disconnect() {
198         Log.d(TAG, "disconnect()");
199         if (mBluetoothAdapter == null || mBluetoothGatt == null) {
200             Log.w(TAG, "BluetoothAdapter not initialized");
201             return;
202         }
203         mBluetoothGatt.disconnect();
204     }
205     /**
206      * After using a given BLE device, the app must call this method to ensure
207      * resources are released properly.
208      */
209     public void close() {
210         Log.d(TAG, "close()");
211         if (mBluetoothGatt == null)
212             return;
213         mBluetoothGatt.close();
214         mBluetoothGatt = null;
215     }
216     /**
217      * Request a read on a given {@code BluetoothGattCharacteristic}.
218      * The read result is reported asynchronously through the
219      * {@code BluetoothGattCallback#onCharacteristicRead(
220      * android.bluetooth.BluetoothGatt, android.bluetooth
221      * .BluetoothGattCharacteristic, int)} callback.
222      *
223      * @param characteristic The characteristic to read from.
224      */
225     public void readCharacteristic(BluetoothGattCharacteristic characteristic) {
226         if (mBluetoothAdapter == null || mBluetoothGatt == null) {
227             Log.w(TAG, "BluetoothAdapter not initialized");
228             return;
229         }
230         mBluetoothGatt.readCharacteristic(characteristic);
231     }
```

```
232
233     /**
234      * Enables or disables notification on a give characteristic.
235      *
236      * @param characteristic Characteristic to act on.
237      * @param enabled If true, enable notification. False otherwise.
238      */
239     public void setCharacteristicNotification(BluetoothGattCharacteristic characteristic,
240                                               boolean enabled) {
241         if (mBluetoothAdapter == null || mBluetoothGatt == null) {
242             Log.w(TAG, "BluetoothAdapter not initialized");
243             return;
244         }
245         mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
246     }
247
248     /**
249      * Retrieves a list of supported GATT services on the connected device.
250      * This should be invoked only after {@code BluetoothGatt#discoverServices()}
251      * completes successfully.
252      *
253      * @return A {@code List} of supported services.
254      */
255     public List<BluetoothGattService> getSupportedGattServices() {
256         if (mBluetoothGatt == null) return null;
257
258         return mBluetoothGatt.getServices();
259     }
260 }
```

Listing A.12: BluetoothLeService.java

A.5.2 SmartphoneServer (Android app)

MainActivity

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = MainActivity.class.getSimpleName();

    private BluetoothManager mBluetoothManager;
    private BluetoothGattServer mBluetoothGattServer;
    private BluetoothLeAdvertiser mBluetoothLeAdvertiser;

    private Set<BluetoothDevice> mRegisteredDevices = new HashSet<>();

    Boolean toggleLight = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mBluetoothManager =
        (BluetoothManager) getSystemService(BLUETOOTH_SERVICE);

        IntentFilter filter =
        new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);

        registerReceiver(mBluetoothReceiver, filter);

        if (!mBluetoothManager.getAdapter().isEnabled()) {

            Intent enableBtIntent =
            new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

            startActivityForResult(enableBtIntent, 0);
        }
        else {
            Log.d(TAG, "Bluetooth enabled...starting services");
            startAdvertising();
            startServer();
        }

        Button toggleButton = findViewById(R.id.toggle_button);
        toggleButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                toggleLight = !toggleLight;
                notifyRegisteredDevices(toggleLight);
            }
        });
    }
```

```
48      /**
49       * Listens for Bluetooth adapter events to enable/disable
50       * advertising and server functionality.
51       */
52      private BroadcastReceiver mBluetoothReceiver = new BroadcastReceiver() {
53          @Override
54          public void onReceive(Context context, Intent intent) {
55              int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
56                                             BluetoothAdapter.STATE_OFF);
57              switch (state) {
58                  case BluetoothAdapter.STATE_ON:
59                      startAdvertising();
60                      startServer();
61                      break;
62                  case BluetoothAdapter.STATE_OFF:
63                      stopServer();
64                      stopAdvertising();
65                      break;
66                  default:
67                      // Do nothing
68              }
69          }
70      };
71      /**
72       * Begin advertising over Bluetooth that this device is connectable
73       * and supports the Remote LED Service.
74       */
75      private void startAdvertising() {
76        Log.d(TAG, "startAdvertising()");
77        BluetoothAdapter bluetoothAdapter = mBluetoothManager.getAdapter();
78        // set the name
79        bluetoothAdapter.setName("S8AlessandroBLE");
80
81        mBluetoothLeAdvertiser = bluetoothAdapter.getBluetoothLeAdvertiser();
82        if (mBluetoothLeAdvertiser == null) {
83            Log.w(TAG, "Failed to create advertiser");
84            return;
85        }
86
87        AdvertiseSettings settings = new AdvertiseSettings.Builder()
88          .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
89          .setConnectable(true)
90          .setTimeout(0)
91          .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
92          .build();
93        AdvertiseData data = new AdvertiseData.Builder()
94          .setIncludeDeviceName(true)
95          .setIncludeTxPowerLevel(false)
96          .addServiceUuid(new ParcelUuid(
97                          RemoteLedProfile.REMOTE_LED_SERVICE))
98          .build();
99
100         mBluetoothLeAdvertiser
101                 .startAdvertising(settings, data, mAdvertiseCallback);
102     }
103
104
```

```
105     /**
106      * Stop Bluetooth advertisements.
107      */
108     private void stopAdvertising() {
109         if (mBluetoothLeAdvertiser == null) return;
110         mBluetoothLeAdvertiser.stopAdvertising(mAdvertiseCallback);
111     }
112
113     /**
114      * Initialize the GATT server instance with the services/characteristics
115      * from the Remote LED Profile.
116      */
117     private void startServer() {
118         Log.d(TAG, "startServer()");
119         mBluetoothGattServer =
120         mBluetoothManager.openGattServer(this, mGattServerCallback);
121
122         if (mBluetoothGattServer == null) {
123             Log.w(TAG, "Unable to create GATT server");
124             return;
125         }
126
127         mBluetoothGattServer
128             .addService(RemoteLedProfile.createRemoteLedService());
129     }
130
131
132     /**
133      * Shut down the GATT server.
134      */
135     private void stopServer() {
136         if (mBluetoothGattServer == null){
137             return;
138         }
139         mBluetoothGattServer.close();
140     }
141
142
143     /**
144      * Callback to receive information about the advertisement process.
145      */
146     private AdvertiseCallback mAdvertiseCallback = new AdvertiseCallback() {
147         @Override
148         public void onStartSuccess(AdvertiseSettings settingsInEffect) {
149             Log.i(TAG, "LE Advertise Started.");
150         }
151
152         @Override
153         public void onStartFailure(int errorCode) {
154             Log.w(TAG, "LE Advertise Failed: "+errorCode);
155         }
156     };
```

```
157     /**
158      * Send a remote led service notification to any devices that are subscribed
159      * to the characteristic.
160      */
161     private void notifyRegisteredDevices(Boolean toggle) {
162       if (mRegisteredDevices.isEmpty()) {
163
164           Log.i(TAG, "No subscribers registered");
165           return;
166       }
167       Log.i(TAG, "Sending update to " + mRegisteredDevices.size() + " subscribers");
168
169       // For each registered device
170       for (BluetoothDevice device : mRegisteredDevices) {
171         BluetoothGattCharacteristic ledDataCharacteristic =
172         mBluetoothGattServer
173                   .getService(RemoteLedProfile.REMOTE_LED_SERVICE)
174                   .getCharacteristic(RemoteLedProfile.REMOTE_LED_DATA);
175          ledDataCharacteristic.setValue(toggle.toString());
176          mBluetoothGattServer.notifyCharacteristicChanged(device,
177                                                           ledDataCharacteristic,
178                                                           false);
179       }
180     }
181     /**
182      * Callback to handle incoming requests to the GATT server.
183      * All read/write requests for characteristics and descriptors are handled here.
184      */
185     private BluetoothGattServerCallback mGattServerCallback =
186     new BluetoothGattServerCallback() {
187
188         @Override
189         public void onConnectionStateChange(BluetoothDevice device,
190                                             int status, int newState) {
191             if (newState == BluetoothProfile.STATE_CONNECTED) {
192                 Log.i(TAG, "BluetoothDevice CONNECTED: " + device);
193                 mRegisteredDevices.add(device);
194             } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
195                 Log.i(TAG, "BluetoothDevice DISCONNECTED: " + device);
196                 //Remove device from any active subscriptions
197                 mRegisteredDevices.remove(device);
198             }
199         }
200
201         @Override
202         public void onCharacteristicReadRequest(BluetoothDevice device,
203           int requestId, int offset, BluetoothGattCharacteristic characteristic) {
204             long now = System.currentTimeMillis();
205             if (RemoteLedProfile.REMOTE_LED_DATA
206                 .equals(characteristic.getUuid())) {
207                 Log.i(TAG, "Read data");
208                 mBluetoothGattServer.sendResponse(device,
209                         requestId,
210                         BluetoothGatt.GATT_SUCCESS,
211                         0,
212                         null);
213             }
```

```
214                 else {
215                   // Invalid characteristic
216                   Log.w(TAG, "Invalid Characteristic Read: " + characteristic.getUuid());
217                   mBluetoothGattServer.sendResponse(device,
218                           requestId,
219                           BluetoothGatt.GATT_FAILURE,
220                           0,
221                           null);
222               }
223           }
224
225           @Override
226           public void onDescriptorReadRequest(BluetoothDevice device, int requestId,
227               int offset, BluetoothGattDescriptor descriptor) {
228                   mBluetoothGattServer.sendResponse(device,
229                       requestId,
230                       BluetoothGatt.GATT_FAILURE,
231                       0,
232                       null);
233           }
234
235           @Override
236           public void onDescriptorWriteRequest(BluetoothDevice device, int requestId,
237                                                BluetoothGattDescriptor descriptor,
238                                                boolean preparedWrite,
239                                                boolean responseNeeded,
240                                                int offset, byte[] value) {
241               if (responseNeeded) {
242                   mBluetoothGattServer.sendResponse(device,
243                           requestId,
244                           BluetoothGatt.GATT_FAILURE,
245                           0,
246                           null);
247               }
248           }
249       };
250
251       @Override
252       protected void onDestroy() {
253           super.onDestroy();
254
255           BluetoothAdapter bluetoothAdapter = mBluetoothManager.getAdapter();
256           if (bluetoothAdapter.isEnabled()) {
257               stopServer();
258               stopAdvertising();
259           }
260
261           unregisterReceiver(mBluetoothReceiver);
262       }
263   }
```

Listing A.13: MainActivity.java

RemoteLedProfile

```
1  public class RemoteLedProfile {
2
3      /* Remote LED Service UUID */
4      public static UUID REMOTE_LED_SERVICE =
5      UUID.fromString ("00001805-0000-1000-8000-00805f9b34fb");
6      /* Remote LED Data Characteristic */
7      public static UUID REMOTE_LED_DATA =
8      UUID.fromString("00002a2b-0000-1000-8000-00805f9b34fb");
9
10     private final static String TAG = RemoteLedProfile.class.getSimpleName();
11
12
13     /**
14      * Return a configured {@link BluetoothGattService} instance
15      * for the Remote LED Service.
16      */
17     public static BluetoothGattService createRemoteLedService() {
18         Log.d(TAG, "createRemoteLedService()");
19
20         BluetoothGattService service =
21         new BluetoothGattService(REMOTE_LED_SERVICE,
22                                  BluetoothGattService.SERVICE_TYPE_PRIMARY);
23
24         BluetoothGattCharacteristic ledData =
25         new BluetoothGattCharacteristic(REMOTE_LED_DATA,
26                 //Read-only characteristic, supports notifications
27                 BluetoothGattCharacteristic.PROPERTY_READ |
28                 BluetoothGattCharacteristic.PROPERTY_NOTIFY,
29                 BluetoothGattCharacteristic.PERMISSION_READ_ENCRYPTED);
30
31         service.addCharacteristic(ledData);
32
33         return service;
34     }
35 }
```

Listing A.14: RemoteLedProfile.java

Bibliography

[1] D. Giusto, A. Iera, G. Morabito, L. Atzori (Eds.), The Internet of Things, Springer, 2010. ISBN: 978-1-4419-1673-0.

[2] L. Atzori et al., The Internet of Things: A survey, Comput. Netw. (2010),

[3] Pico i.MX7 Development Kit for Android Things Hardware Manual, REV B1.

[4] Azzola, Francesco. (2017), *Android Things Projects – Effeciently build IoT projects with Android Things*, Packt Publishing.

[5] Sławomir Jasek. *GATTACKING BLUETOOTH SMART DEVICES*, SecuRing.

[6] Wondimu K. Zegeye, *Exploiting Bluetooth Low Energy Pairing Vulnerability in Telemedicine*.

Sitography

[7] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016 https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-\in-use-in-2017-up-31-percent-from-2016

[8] Supported hardware for Android Things. https://developer.android.com/things/hardware

[9] NXP i.MX7D starter kit. https://androidthings.withgoogle.com/#!/kits/starter-kit

[10] NXP i.MX7D platform. https://developer.android.com/things/hardware/imx7d.html

[11] Platform Differences https://developer.android.com/things/get-started/platform-differences

[12] Content Providers https://developer.android.com/guide/topics/providers/content-providers

[13] ANR - Application Not Responding https://developer.android.com/topic/performance/vitals/anr

[14] PWM Arduino https://www.arduino.cc/en/Tutorial/PWM

[15] I2C Android Things https://developer.android.com/things/sdk/pio/i2c

[16] User-space drivers https://developer.android.com/things/sdk/drivers

[17] LiveData class https://developer.android.com/topic/libraries/architecture/livedata

[18] God Object in OOP https://en.wikipedia.org/wiki/God_object

[19] CloseResource checker https://static.juliasoft.com/docs/latest/CloseResource.html

[20] GDPR checker https://static.juliasoft.com/docs/latest/Gdpr.html

[21] Bluetooth Low Energy https://en.wikipedia.org/wiki/Bluetooth_Low_Energy

[22] Bluetooth Low Energy overview https://developer.android.com/guide/topics/connectivity/bluetooth-le

[23] Bluetooth in Android Things https://developer.android.com/things/sdk/apis/bluetooth

[24] The BluetoothConnectionManager class https://developer.android.com/reference/com/google/android/things/bluetooth/BluetoothConnectionManager.html

[25] BLE master/slave, GATT client/server, and data RX/TX basics https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2015/08/06/_reference_ble_mas-gviy