Nosakhare Osamwonyi

Development of a Bluetooth Low Energy Educational Framework

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree porgramme In Information Technology

Bachelor's Thesis

25 April 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Nosakhare Osamwonyi Development of a Bluetooth Low Energy Education Framework 49 pages 25 April 2016
Degree	Bachelor of Engineering
Degree Programme	Degree porgramme In Information Technology
Specialisation option	Software Engineering
Instructor(s)	Anssi Ikonen, Head of Degree Programme

This project consisted of two tasks. The first one was to provide a theoretical documentation of the Bluetooth Low Energy (BLE) device whose first generational versions emerged in 2010. The theoretical documentation basically covers the protocol stack of the device, which consists of the Controller, Host and Application layer, all of which govern how communication should ensue. The controller layer is the point of BLE to BLE interface, while the host layers have to do with the internal logics, security management, data storage and manipulation, client and server role of the device. The application layer is the user to BLE interface.

The second task was to create an Android mobile application to establish a communication link with the BLE peripherals. The specimen BLE module used in this project was the serial Bluetooth 4.0 module from the vendor TinySine, which was supported on the Arduino platform. The generic attribute GATT of the BLE is responsible for data storage, and to access the GATT from application programs the service and characteristic UUID is required.

This is a framework for educational purposes. The future trend is that the BLE peripheral used in this project would be replaced by BLE sensors in similar projects, establishing communication links with a central device and reporting their respective measurements to them.

Keywords	BLE, L2CAP,GATT, GAP, SMP, ATT, LL, LE,UI

Contents

1 Introduction

A recent Bluetooth technology has emerged called the Bluetooth Low Energy (BLE), and different versions of this device have been in the market since the year 2010.

The goal of this project was to develop a framework for the Bluetooth Low Energy for educational purposes by studying and documenting a comprehensive theoretical background of the BLE, and developing an Android mobile application to establish a communication link with the BLE peripherals in a simple piconet.

This project was done under the auspices of the department of Information Technology Helsinki Metropolia University of Applied Sciences. As the project title connotes “The Development of Bluetooth Low Energy Educational Framework”, the purpose is to present a comprehensive framework to serve educational purposes, that is, to be used for learning and teaching.

2 Bluetooth Low Energy Device

The Bluetooth Low Energy (BLE) is also commonly called by the name Bluetooth Smart belonging to the Bluetooth 4.0 Core Specification. Though BLE has a common feature as the Bluetooth classic, its design goals and lineage totally differ from the classic Bluetooth. Wibree was the former name of BLE when first created by Nokia, before it was adopted by the Bluetooth Special Interest Group (SIG). A radio standard with a minimum power consumption, low bandwidth, low power and low complexity, was the main focus of the inventors when BLE was first invented. [1,15.]

2.1 Configurations

The Classic Bluetooth and the BLE are both under a common Bluetooth specification. Despite this unifying specification, there is non-existence of direct compatibility for both Bluetooth standards. The inability for the two technologies to communicate is hinged on their differences in the upper protocol layers, the applications and the on-air protocol. [1,17.]

Table 1. Specification configuration. Reprinted from Townsend [1,18]

Device	classic Bluetooth (BR/EDR) support	BLE support
Pre-4.0 Bluetooth	Yes	No
4.x Single-Mode (Bluetooth Smart)	No	Yes
4.x Dual-Mode (Bluetooth Smart Ready)	Yes	Yes

Table 1 explanations is that Bluetooth versions prior to the emergence of version 4.0, have a Basic Rate (BR) or Enhance Data Rate (EDR) support, while BLE support excluded.

Then for versions 4.x (x can have any integer values starting from 0) Single-Mode (Bluetooth Smart) has only BLE support, the classic Bluetooth excluded. The implication is that this device implementing BLE is compatible with both the single-mode and dual mode devices.

The device versions 4.x (x can have any integer values starting from 0) Dual-Mode is compatible with any Bluetooth device, that is the classic Bluetooth and BLE inclusive.

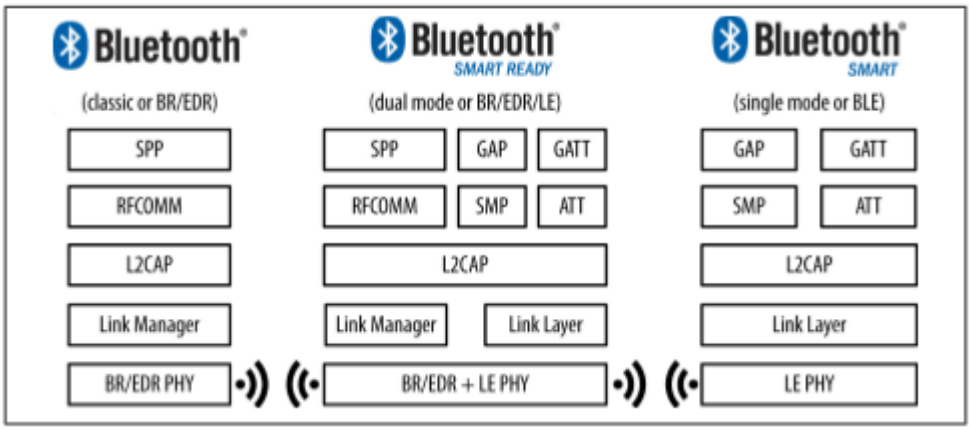

Figure 1. Configuration between Bluetooth versions and device types. Copied from Townsend (2015) [1,18]

In figure 1, the dual mode Bluetooth Smart Ready is central to the single mode or BLE device and the classic or BR/EDR device. Communication is established in the physical (PHY) layers of the devices, with only the dual mode standard having the capability of communicating with the BLE and classic Bluetooth standards. [1,18.]

2.2 BLE Stack

The building blocks of BLE as any other Bluetooth devices are Application layer, Host layer, and the Controller layer.

- APPLICATION Layer this is made up of the user application interface with the Bluetooth protocol stack.
- HOST Layer comprises the upper layers of the Bluetooth protocol stack.
- CONTROLLER Layer is constituted of the lower layers of the Bluetooth protocol stack and the radio.

[1,19.]

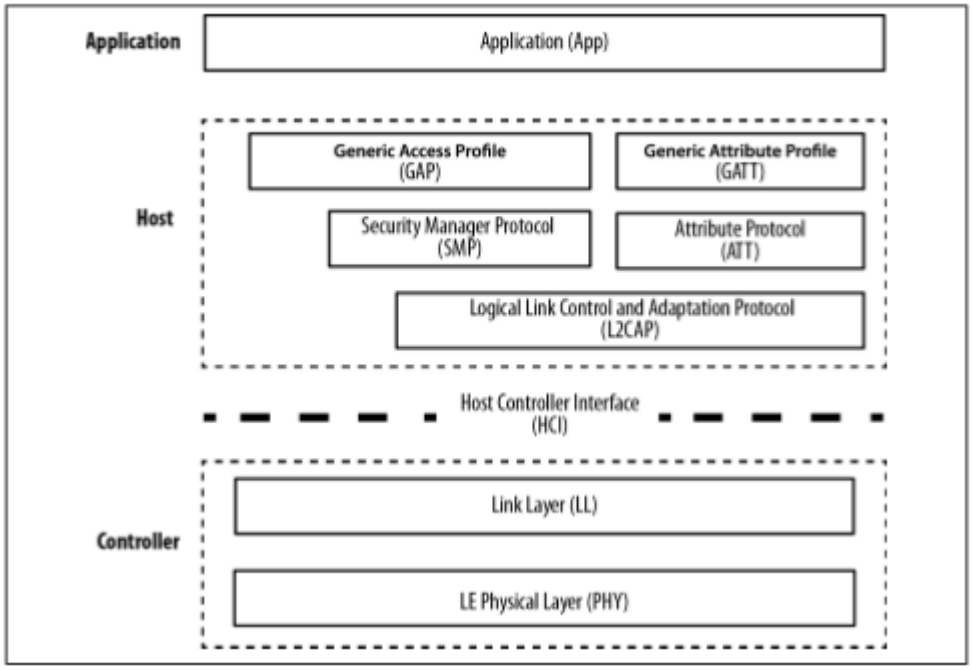

Figure 2. A single-mode BLE protocol stack. Reprinted from Townsend (2015) [1,30]

According to figure 2, the controller is made up of the Link Layer (LL) and the Physical layer (PHY). Connection state, scanning, advertising, sequence and timing of transmitted and received frames are all handled by the LL, while the physical layer has to do with the BLE Radio. In general, the controller manages physical layer packets and all related timing. In between the host and the controller lies the Host Controller Interface (HCI), interfacing the controller from the Host. The Logical Link Control and Adaptation layer Protocol (L2CAP) is a multiplexer for combining and channeling streams of data

between the Controller and the Host elements, which includes the Generic Access Profile (GAP), Security Manger (SM) and, Generic Attribute Profile (GATT). The Security Manager (SM) manages all security operations, such as encryption and authentication mechanism, key allocation and pairing. Generic Attribute Profile (GATT) provides the framework required for services discovery, reading and writing characteristics values on paired devices. Generic Access Profile (GAP) starts, sets up and manages the connection with other Bluetooth devices, provides a means for application configuration and enables different modes of operation. [2.]

2.3 Physical Layer (PHY)

The Physical Layer is the lowest layer that interfaces with other BLE device, and its features are the analog circuits, required to modulate and demodulate analog signals, and transforming these signals into digital symbols at transmission or reception. The frequency band used by the physical layer in BLE devices is the ISM (Industrial, Scientific, and Medical) band frequency for the radio communication which is of 2.4 GHz. The ISM band is divided into 40 channels, ranging from 2.4000 GHz to 2.4835 GHz. Physical Layer operates on three channels 37, 38, 39 for advertising channels to set up connections and send broadcast data, while the remaining 37 channels are used for connection data. [1,30.]

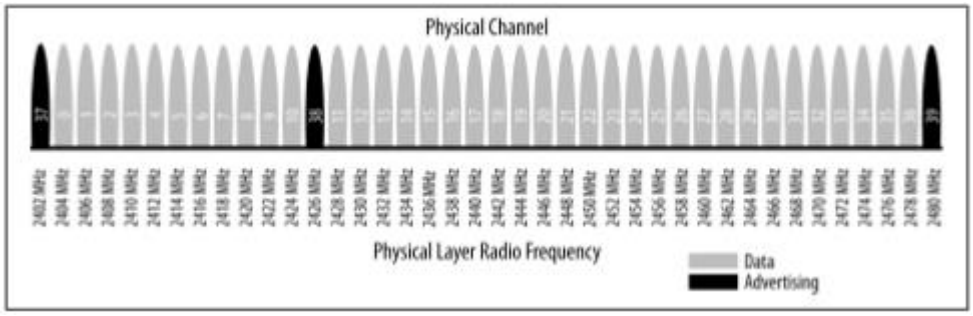

Figure 3. Frequency channels. Reprinted from Townsend (2015) [1,31]

Figure 3 describes the physical channels and their various frequency levels, with the light colored channels representing data channels, while the dark colored one is for advertising. [1,30.]

2.4 Link Layer

In the controller layer the link layer is directly an interface to the PHY and its implementation is a mixture of software and custom hardware. The Host controller Interface (HCI) is the standard interface that isolates the link layer from the higher layers due to its timing requirements and complexity.

To avoid overloading the CPU that implements all of the software layers in the stack, the hardware part of the link layer is characterized with the following functionalities:

- Random number generation
- Data whitening
- Cyclic Redundancy Check (CRC) generation and verification
- Advance Encryption Standard (AES) encryption
- Preamble, Access Address, and air protocol framing.

The link state of the radio signal is managed by the software part of the link layer. Depending on the application requirements and use-case, a BLE device can function as a master only, a slave only, or both. A slave device is that which advertises its presence and accepts connections, while a master device is that which initiates a connection.

The connection possibilities are that a master can establish connections with multiple slaves, and conversely a slave can establish a connection with multiple masters. In most set-up tablets, smart-phones or any related devices typically function as the master, while the devices such as temperature sensors act as a slave.

The roles defined by the link layer include:

- Advertiser is a device that sends advertising packets.
- Master is a device that initiates and manages a connection.
- Slave is a device that accepts a connection request and follows the master’s timing.

- Scanner is a device scanning for advertising packets.

These roles can be logically grouped into two pairs: advertiser and scanner (when not in an active connection) and master and slave (when in a connection).

The groups consisting of two pairs can be further derived from the listed roles:

- Advertising and Scanner when a peer device does not possess an active connection
- Master and Slave when a peer device is in an active connection.

[1,32.]

The link layer uses a Bluetooth device address to identify Bluetooth devices among peers. The Bluetooth device address is a 6-byte number and two types of this address exist, which are:

- Public Device Address a permanent address that does not change throughout the device lifetime, but it must be registered with the IEEE Registration Authorities.
- Random Device Address is an address generated at runtime or preprogrammed on the device.

The link layer is also responsible for advertising and scanning operations which are required in establishing a peer to peer connection. In BLE protocol stack implementation these operations are simplified compared to the classical Bluetooth. The simplified advertising packet does broadcast data for applications which do not require an overhead of a full connection setup, and to discover and establish a connection with a slave device.

The components of an advertising packet are advertising data payload, and header information comprising of Bluetooth device address. An advertiser need not know of the presence of any scanning device when broadcasting packets over the air. The advertising interval ranges from 20 ms to 10.24 s, and packets are broadcast at higher frequencies for shorter intervals. It is true that power consumption is higher for higher number of packets transmitted. [1,33.]

Figure 4 depicts that advertising is done using three channels, and advertising packets are received successfully by a scanner only when they overlap randomly.

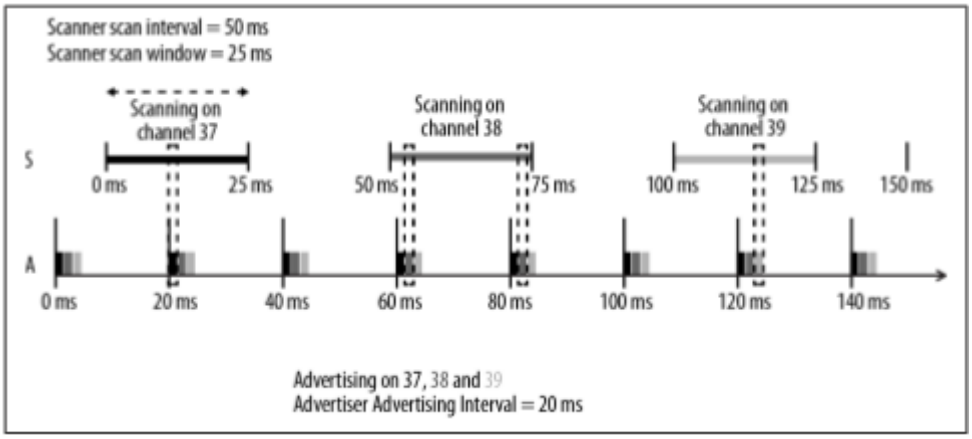

Figure 4. Advertising and Scanning. Reprinted from Townsend (2015) [1,34]

The two parameters, scan interval and scan windows respectively define how frequently and for how long a scanner device listens to advertising packets. Note that the amount of time the radio must be on has a great impact on power consumption. Now for scanning, two fundamental types have been defined by Bluetooth specification, and this are: passive scanning and active scanning. In passive scanning, a scanner listens to advertising packets, and an advertiser has no knowledge of the number of packets (either one or more packets) a scanner has received. In active scanning, a Scan Request (SR) is sent to the advertiser by the scanner when it receives an advertising packet. Upon the reception of the SR by the advertiser, it will send back a Scan Response.

The difference between a passive and active scanning is illustrated by figure 5.

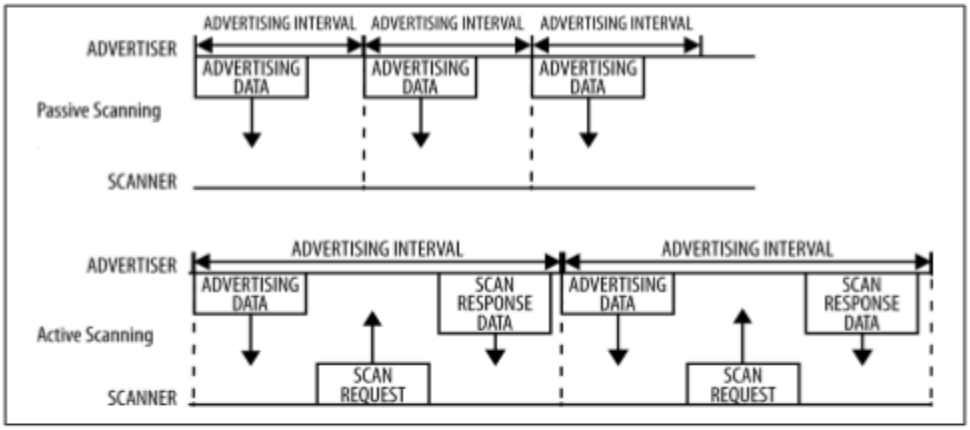

Figure 5 Passive and Active scanning. Copied from Townsend (2015) [5,34]

[1,35.]

In setting up a connection on the link layer a master device scans to search for advertisers that are accepting connection requests. Filtering of the advertising packets can be done using advertising data or by using a Bluetooth address. The master sends a connection request to a suitable advertising slave upon its detection, and a connection is established when the slave responds to the request.

A connection is a sequence of data exchanges between the slave and the master at predefined times. Shown in Figure 6, each exchange is called a connection event.

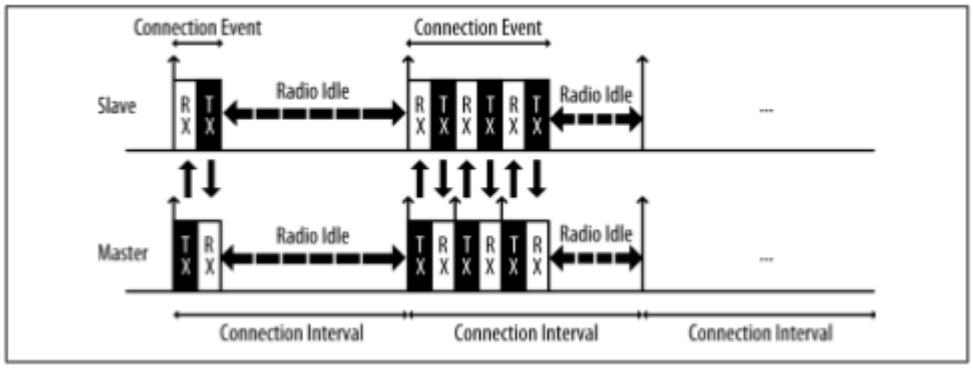

Figure 6 Connection events. Copied from Townsend (2015) [1,36]

The link layer is responsible also for changing the connection parameters, and encryption of data, in addition to advertising, scanning, establishing connections, and transmitting and receiving data. [1,37.]

2.5 Host Controller Interface

The HCI is the interface standing between a host and a controller. It is responsible for the serial communication that exists between the host and the controller. The hard real-time requirements of the BLE protocol stack is implemented only by the controller and that makes it practical to separate it from the host which is implementing a complex and a less timing-stringent protocol stack by an advance CPU.

In most smart phones, personal computers and tablets, stack protocol configuration is such that the host and application operations are in the main CPU, while the controller can be found on a separate chip connected through a USB or UART.

Furthermore, the host and the controller interact through the HCI sets of commands and events, along with a data packet format and a set of rules for flow control and procedures, based on the Bluetooth specification. [1,38.]

3 Host Layer

The host layer is made up of the Logic Link Control and Adaptation Protocol (L2CAP), the Attribute Protocol Layer (ATT), the Security Manager Protocol (SMP), the Generic Acces Profile (GAP), and the Generic Attribute profile (GATT).

3.1 Logic Link Control and Adaptation Protocol

The L2CAP serves two main functions, as a protocol multiplexer, and for performing fragmentation and recombination.

In protocol multiplexer operation, L2CAP encapsulates multiple protocols from upper layers into the BLE packet format, and also breaks down or reverts the BLE packet format to the upper layers' standard. In fragmentation L2CAP breaks down the large packets from the upper layers into smaller bytes that fit the 27-byte maximum payload size required at the transmission end. On the other hand, at the receiving end, L2CAP performs recombination, in which it recombines the received multiple packet fragments into a single packet, and sends them to the upper layers of the protocol.

The two main protocols that L2CAP layer routes to are: the Security Manager Protocol (SMP) and the Attribute Protocol (ATT).
[1,39.]

3.2 Attribute Protocol

The Attribute Protocol (ATT) is referred to as a stateless protocol because it can either act as a server or a client, depending on the attributes presented by a device. Irrespective of BLE device being a master or a slave, it can operate as a client, a server or as both a server and a client.

Sequencing operations are strict in ATT protocol, such that when a request is sent, it will be impossible to send a further request, until after a response to the first request has been received and processed. This behavior is applicable to both directions, for two peers acting both as a server and a client. Data in ATT protocol servers are orga-

nized as attributes, with each assigned a 16-bit attribute handle, a universally unique identifier (UUID), a set of permissions, and a value.

An attribute handle is an identifier, required to access an attribute value. The nature and type of the data present in the value is specified by the UUID. During a write operation by a client to a server, it is essential to provide data consistent with the servers' attribute type. Otherwise the operation will be rejected. In read operation the client understands the data type, by analyzing the UUID of the attribute from the server device. [1,40.]

3.3 Security Manager

The security manager (SM) architecture allows the protocol to act as both a protocol and as security algorithms, and enabling the protocol stack to generate security keys that are exchanged among peers. The keys are required for secure communication over an encrypted link, and to keep in secret the Bluetooth address from a malicious attacker. Two roles defined in SM are an initiator and a responder. The initiator's role corresponds to the master link layer and a responder always corresponds to the slave link layer. The initiator usually initiates the start of a procedure in most cases, although the responder can request the beginning of a procedure asynchronously. [1,42.]

3.4 Generic Attribute Profile

The generic attribute (GATT) profile is an important aspect of the BLE protocol stack. It is required for transfer of data, and for the definition of how data are organized and how they are exchanged between applications. GATT data is encapsulated in a service, which is made up of a single or multiple characteristics. [1,46.]

GATT is mainly involved in the exchange of all profile and user data over a BLE connection, with its operations limited to only data transfer procedures and formats. All standard BLE profiles (device) must be GATT-compliant which means the application

and user data are formatted, packed and sent based on GATT definitions. This ensures interoperability between the BLE devices from different vendors.

According to the Bluetooth specification GATT defines the roles which are applied by interacting devices. These roles are a client and a server. In the client mode a BLE device sends a request to a server and the server replies the client request with a response. In addition to the response, in some instances the server sends its initiated update to the client. [1,65.]. In the server mode the device sends a response to a client upon receiving a request from it, and a server can also be configured to send updates it has initiated, to a client. Furthermore, it stores user data in attributes, and renders this data to the client adequately.

3.4.1 Universally Unique Identifier (UUID)

Bluetooth specification defines the Universally Unique Identifier (UUID) which is a 16-byte number that is guaranteed to be globally unique. Additionally, Bluetooth specification defines two more shortened UUID formats and they are: a 16-bit and a 32-bit UUIDs. The reason for this short form is to pack the UUIDs in such a way that it fits into the 27-byte data length of the link layer, because the original UUID of 16 bytes would take a large chunk of the data payload length of the link layer. The shortened version of the UUID is reconstructed into the original 16-byte (128-bit) UUID, by inserting the 16-bit or 32-bit short value into the Bluetooth Base UUID, for example:

xxxxxxxx-0000-1000-8000-00805F9B34FB

The first 8-digits (as represented by x), including the leading zeroes of the above UUID indicates a 16- or 32-bit short value. Note that shortening is not available for all UUIDs, and an example is called a vendor-specific UUID. It lacks the shortening feature, because it does not derive from the Bluetooth Base UUID. For this reason the 128-bit UUID values are always used. Although the Special Interest Group (SIG) defines and specifies UUIDs for all types, services, and profiles, it is possible for an application developer to generate its own UUID based on the specific use case required by an application which have not been provided by in the SIG specifications. [1,66.]

3.4.2 Attribute and Data Hierarchy of a GATT

An attribute and data hierarchy feature in GATT, enables the organization of attributes, so that it is practically re-usable, and information exchange between the client and server are made to obey a set of rules that collectively constitute the framework used by all GATT-based profiles. Figure 7 describes the data hierarchy introduced by GATT. [1,70].

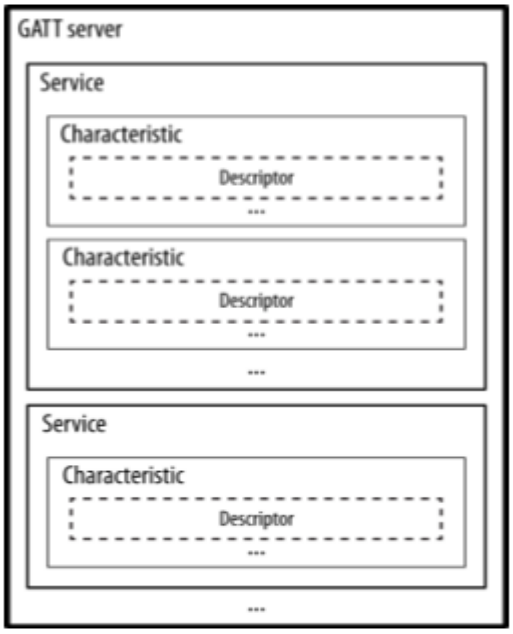

Figure 7. GATT data hierarchy. Reprinted from Townsend (2015) [1,71]

According to figure 7, a GATT server contains attributes that are grouped as services and each of the service can have a zero, one or more characteristics. A characteristic on the other hand can contain a zero, one or more descriptors. Data exchange between BLE depends on these GATT hierarchical attributes, and for all BLE devices to be GATT compatible, it must strictly enforce this hierarchy. [1,71.]

The GATT services group similar attributes in a common segment of the attribute information set in the GATT server. Attributes within a single service are known as service definition, and each of these service definitions starts with a service declaration that denotes the beginning of a service.

Table 2 below is an example of a service declaration attribute showing its attribute type and value format.

Table 2. Service Declaration attribute. Copied from Townsend (2015) [1,72]

Handle	Type	Permissions	Value	Value length
0xNNNN	UUID$primary\ service$ (0x2800) or UUID$secondary\ service$(0x2801)	Read Only	Service UUID	2, 4, or 16 bytes

UUID$primary\ service$ (0x2800) and UUID$secondary\ service$(0x2801) are SIG UUID standards, which are exclusively used to introduce a service. UUID$secondary\ service$ serves as a modifier for the UUID$primary\ service$, and carries no meaning on its own. UUID$secondary\ service$ are hardly use in practice. Permissions are read only, which means the service cannot be written to. [1,72.]

The characteristic in GATT service is an entity that contains the user data which contain characteristic declaration and the characteristic value. Characteristic declaration holds the metadata that gives information about the user data, while Characteristic value holds the actual user data. Sometimes the descriptor can come after the characteristic value. The declaration, value and descriptors are collectively known as a bundle of attributes that are constituents of a single characteristic.

The Characteristic declaration attribute marks the beginning of a characteristics, and according to the SIG, the standardized UUID type for the characteristic declaration attribute UUID is (0x2803).[1,73.]. Every operation and procedure that is implemented with the GATT characteristic can be encoded here on the characteristic properties which is 8-bit long. The characteristic properties that can be encoded in this bit-field are listed in table 3.

Table 3. Characteristic properties. Reprinted from Townsend (2015) [1,74]

Property	Location	Description
Broadcast	Properties	If set, allows this characteristic value to be placed in advertising packets, using the Service Data AD Type
Read	Properties	If set, allows clients to read this characteristic using any of the ATT read operations listed in “ATT operations”
Write without response	Properties	If set, allows clients to use the Write Command ATT operation on this characteristic
Write	Properties	If set, allows clients to use the Write Request/Response ATT operation on this characteristic
Notify	Properties	If set, allows the server to use the Handle Value Notification ATT operation on this characteristic
Indicate	Properties	If set, allows the server to use the Handle Value Indication/Confirmation ATT operation on this characteristic
Signed Write Command	Properties	If set, allows clients to use the Signed Write Command ATT operation on this characteristic
Queued Write	Extended Properties	If set, allows clients to use the Queued Writes ATT operations on this characteristic
Writable Auxiliaries	Extended Properties	If set, a client can write to the descriptor described in “Characteristic User Description Descriptor”

Usually a client is required to read these properties. From there it knows what operations it is permitted to execute on the characteristics. [1,74.] The essential elements of a characteristic are value handle and descriptor. Characteristic Value Handle is 2-bytes (16-bit) long, and has the attribute handle of the attribute that holds the actual value of the characteristic. Characteristic Descriptor is used to provide additional information about the characteristic and its value. It comes after the characteristic value attribute which is within the characteristic definition. The only attribute that is available in the

descriptor is its characteristic descriptor declaration, and it has its own descriptor type UUID. [1,75.]

3.5 Generic Access Profile

GAP defines the BLE uppermost control layer, and dictates how devices carryout device discovery, connection, security establishment, in order to guarantee interoperability and permit data flow between BLE devices from different vendors. For GAP to perform its operations of controlling devices inter-action at a lower level, it defines the following device interactions in accordance with the BLE core specification and they include: roles, modes, procedures, security, and additional GAP data format.

With roles defined by GAP, BLE devices have the capability to perform one or several roles simultaneously. Each and every role has its own conditions and behaviors. Two or more roles can be combined to establish the needed requirements for devices communication. The roles a BLE device can adopt are: as a broadcaster, as an observer, as a central, and as a peripheral.

In modes the BLE devices can switch to different existing states in order to perform a certain task and procedure. Switching modes can be triggered by a user interface actions or automatically when required. In order for a device to attain a certain goal, it needs to follow a sequence of actions, which is known as a procedure.

GAP defines security modes and procedures that specify how peers set the level of security required by a particular data exchange and later how that security level is enforced. Additional GAP Data Format is also used as a placeholder for certain additional data format definitions that are related to the modes and procedures defined by the GAP specification. It should be noted that a BLE device can adopt one or combination of the GAP roles, without any restrictions. [1, 49-51]

4 Establishing Communication Link between Android Mobile Application and the Serial Bluetooth 4.0 BLE Peripheral.

These aspect has to do with the upper layer or the application layer of the BLE protocol stack, were programming of Bluetooth devices are done. This chapter presents the development of a Bluetooth Android mobile application, and subsequently establishing of a communication link between the mobile application and the BLE peripheral devices. The communication link is a wireless one established through a Bluetooth network formed between the mobile application and the Bluetooth BLE 4.0 peripheral. The main goal of this implementation is to set up a Bluetooth piconet network in which a mobile device is acting as the central, while the BLE peripherals are slaves, and to show that the peripherals are able to send text data to the central device.

Keep in mind that the communication between the BLE peripheral and the mobile application can be described as a client-server communication as earlier described in chapter 2. When a generic attribute (GATT) characteristic is writing data to another device, it is acting as a server, while on the other hand when it is receiving data it is said to be acting as a client.

4.1 Android Mobile Application

The Android mobile app developed in this project has been named the MetroGattBle, a Bluetooth application established for the purpose of receiving data from multiple BLE peripherals (a maximum of seven peripherals) in the network. It is capable of scanning for BLE peripherals around its vicinity, making connections with the Bluetooth devices depending on the number of devices user selects on the application, and then reading text data from the connected BLE devices. A minor requirement but not the main focus is that the application can also send data to multiple BLEs on the network.

There are two Independent Development Environments (IDE) that exist for developing an Android mobile application: the Eclipse IDE and the Android studio. The android studio IDE was used in the development of the mobile application in this project.

The engine room of this mobile application as most other android applications, is in the Java source folder containing the components .java files, and for this application they are the:

- MetroBluetoothLeService class
- MetroDeviceControlActivity class
- MetroDeviceScanActivity class and,
- MetroSampleGattAttributes class.

The sequence diagram show the interaction between the above components, as shown in figure 8.

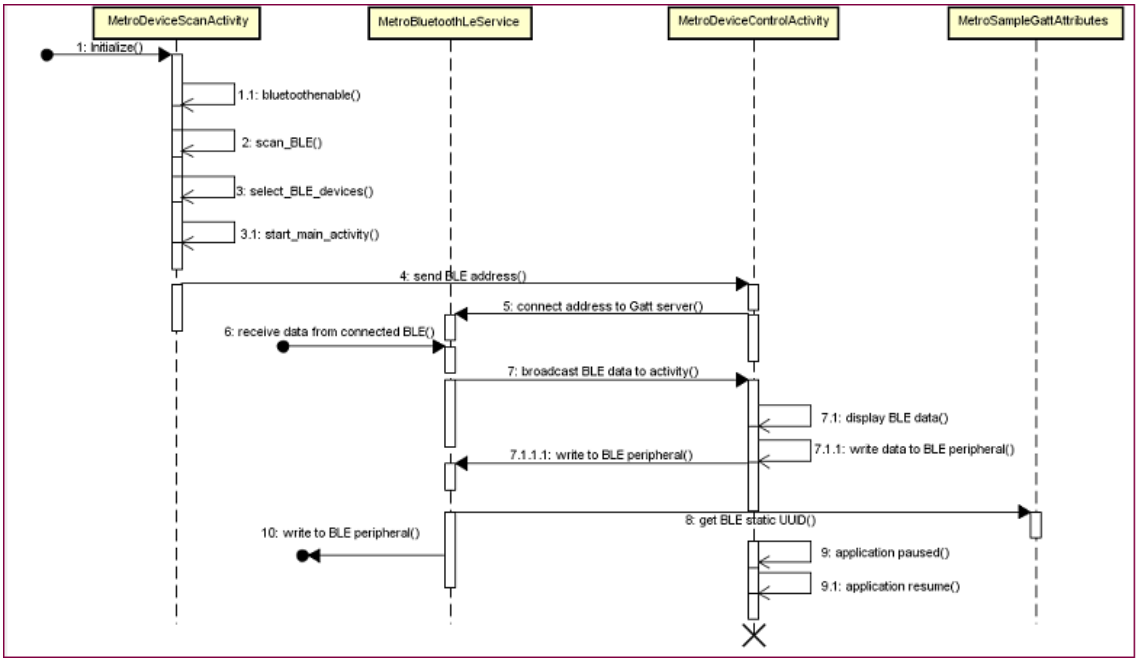

Figure 8. Inter-action diagram between the components of the application.

In figure 8, the first component a user encounters when the application is launched, is the MetroDeviceScanActivity which is a type of activity a user can interact with. The MetroBluetoothLeService component runs on the background and interface the mobile application with the peripherals. The MetroDeviceControlActivity is the main activity in which users of the application can edit and send text to a BLE peripheral. The MetroSampleGattAttributes component is used to store the known UUIDs of the BLE peripherals.

4.1.1 MetroBluetoothLeService

The MetroBluetoothLeService is a class that extends the service class which is a base class in android. A service is an application component that performs operations in the

background, which implies it does not provide a user interface. The service allows inter-process communication (IPC) with other components; for example playing music on the mobile can be done on the background.

The MetroBluetoothLeService class diagram is shown by figure 9 below.

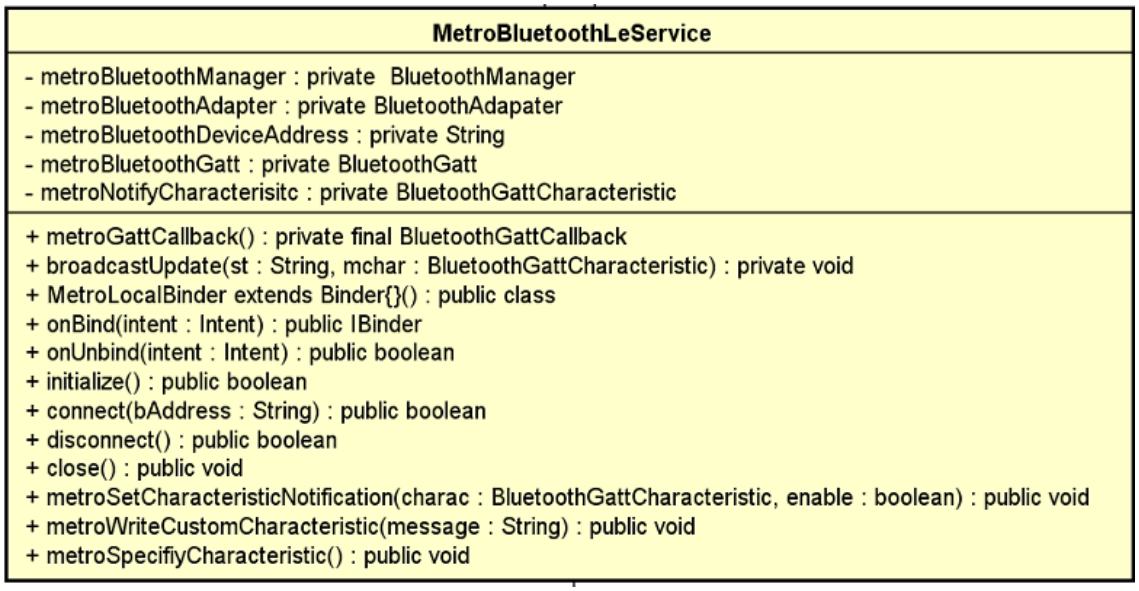

Figure 9 class diagram of the MetroBluetoothLeService showing essential attributes and methods.

The figure shows the essential attributes and methods of this class. The BluetoothManager is a manager that performs all Bluetooth management and for enabling the application get BluetoothAdapter instance. [7]. The BluetoothAdapter instance provides the means for the application to perform device discovery, get the list of devices bonded to the application, obtain an instance of a BluetoothDevice, scan for BLE devices, and listen to a connection request sent from another device. [8]. The BluetoothGatt instance provides the required functionality to have an effective communication with a BLE peripheral. [9]. The BluetoothGattCharacteristic is a class whose instance is a GATT characteristic of a BLE device, and the characteristic is the fundamental data element. [10]. The basic role of the service component in the application is providing the methods or means of: connecting the application to GATT server, mobile application communicating and monitoring BLE peripheral characteristic, broadcasting data within the application, writing data to a GATT Server.

By connecting the application to BLE GATT server we obtain an instance of the *BluetoothGatt* class object, to bind with the *BluetoothGattCallback* object an interface class for monitoring changes in the characteristic of the BLE device.

```
public boolean connect(final String address)
{

    if (metroBluetoothAdapter == null || address == null) {
        return false;
    }

    if (metroBluetoothDeviceAddress != null &&
        address.equals(metroBluetoothDeviceAddress)
            && metroBluetoothGatt != null) {
        if (metroBluetoothGatt.connect()) {
            metroConnectionState = METRO_STATE_CONNECTING;
            return true;
        } else {
            return false;
        }
    }

    final BluetoothDevice device =
    metroBluetoothAdapter.getRemoteDevice(address);
    if (device == null) {
        return false;
    }

    metroBluetoothGatt = device.connectGatt(this, false,
                         metroGattCallback);
                         metroBluetoothDeviceAddress = address;
    metroConnectionState = Metro_STATE_CONNECTING;
    return true;
}
```

Listing 1. Connecting the app to the Gatt server of the BLE device.

This connection establishes the communication interface for monitoring and receiving data from BLE characterisitic. In order for the application to be capable of monitoring changes on the characteristic, it is essential to set or enable the notification property of the characteristic by using the *BluetoothGatt setCharateristicNotification* method.
The string parameter is the Bluetooth address of the BLE device that the application uses to set up the GATT connection.

To be able to communicate and monitor changes of the BLE peripheral characteristic, the mobile application needs to implement the *BluetoothGatt* callbacks, by instantiating a *BluetoothGattCallback*, an abstract class containing the callback methods. The block

of code below is an instance of the *BluetoothGattCallback*, enabling the interaction between a BLE peripheral and the mobile application.

```
{
   @Override
   public void onConnectionStateChange(BluetoothGatt metrogatt,
                                       int metrostatus,
                                        int metronewState)
   {
      String metrointentAction;
      if (metronewState == BluetoothProfile.METRO_STATE_CONNECTED)
{
           metrointentAction = METRO_ACTION_GATT_CONNECTED;
           metroConnectionState = METRO_STATE_CONNECTED;
           broadcastUpdate(metrointentAction);
      }

     else if(metronewState==
             BluetoothProfile.METRO_STATE_DISCONNECTED)
     {
           metrointentAction = METRO_ACTION_GATT_DISCONNECTED;
           metroConnectionState = METRO_STATE_DISCONNECTED;
           broadcastUpdate(metrointentAction);
     }
   }

   @Override
   public void onServicesDiscovered(BluetoothGatt metrogatt,
                                    int metrostatus)
   {
     if (metrostatus == BluetoothGatt.GATT_SUCCESS) {
           broadcastUpdate(METRO_ACTION_GATT_SERVICES_DISCOVERED);
     }
   }
```

```
@Override
public void onCharacteristicRead(BluetoothGatt metrogatt,
                                 BluetoothGattCharacteristic
                                 metrocharacteristic, int
                                 metrostatus)
{
   if (metrostatus == BluetoothGatt.GATT_SUCCESS)
   {
       broadcastUpdate(METRO_ACTION_DATA_AVAILABLE,
       metrocharacteristic);

   }
}

@Override
public void onCharacteristicWrite(BluetoothGatt metrogatt,
                        BluetoothGattCharacteristic
                        metrocharacteristic, int metrostatus)
{

}

@Override
public void onCharacteristicChanged(BluetoothGatt metrogatt,
                                    BluetoothGattCharacteristic
                                    metrocharacteristic)
{
        broadcastUpdate(METRO_ACTION_DATA_AVAILABLE,
                        metrocharacteristic);

}
```

Listing 2. Instance of a BluetoothGattCallback block with its callback methods.

The *onConnectionStateChange* method is a method for keeping track of the state of connection, whether a client is connected to or disconnected from a server. The *onServicesDiscovered* method is a method that monitors, and on noticing an update in the list of services, characteristics and descriptor, it triggers. The *onCharacteristicRead* method is a method from which the application obtains the result of a characteristic read operation when this method is invoked. The *onCharacteristicWrite* method here the outcome of a characteristic write operation is presented by this method callback.
The *onCharacteristicChanged* method an occurrence of a characteristic notification triggers this callback method. [11.]

When the application receives data from a connected BLE characteristic, through the *BluetoothGattcallback* instance, it is necessary to broadcast this data within the application; for instance the data needs to be visualized at the user interface. The *broad-*

castUpdate method in the service performs the broadcasting role by obtaining the data from a characteristic.

```
private void broadcastUpdate(final String metroaction,
                             final BluetoothGattCharacteristic
                             metrocharacteristic)
{
      final Intent intent = new Intent(metroaction);
      final byte[] data = metrocharacteristic.getValue();
      if (data != null && data.length > 0) {
          final StringBuilder stringBuilder =
          new StringBuilder(data.length);
          for(byte byteChar : data) {
              stringBuilder.append(String.format("%02X",
              byteChar));

          }
          intent.putExtra(EXTRA_DATA, new String(data));

      }

  sendBroadcast(intent);
}
```

Listing 3. The broadcastUpdate method for sending data within the mobile app.

Listing 3 shows that the method has a *BluetoothGattCharacteristic* parameter which enables this method to retrieve data from any characteristic server communicating with the application at any particular time. The android *sendBroadcast* method, called within this *broadcastUpdate* method, does the actual broadcasting by using an intent.

To write an associated characteristic of a BLE peripheral the central device (the mobile application) needs to know its Service UUID and characteristic UUID. The UUIDs of the TinySine Serial Bluetooth 4.0 BLE Module used in this project was obtained by using the BLE scanner (a Google app for probing GATT service in BLE) application to scan the active peripherals. The Service and Characteristic UUID are:

```
Service UUID =   "0000ffe0-0000-1000-8000-00805f9b34fb"
Characteristic UUID = "0000ffe1-0000-1000-8000-00805f9b34fb"
```

Listing 4. UUIDs of service and a characteristic of the BLE module.

The mobile application requires the UUIDs for sending data to the GATT client characteristic.

```
public void metrowriteCustomCharacteristic(final String message)
{

   BluetoothGattService metroCustomService =
   metroBluetoothGatt.getService
   (UUID.fromString
   (SampleGattAttributes.TINYSINE_CUSTOM_SERVICE_UUID));

   BluetoothGattCharacteristic metroWriteCharacteristic =
   metroCustomService.getCharacteristic
   (UUID.fromString
   (SampleGattAttributes.TINYSINE_CUSTOM_CHARACTERISTIC_UUID));

   if (message.length() > 0)
   {
       byte[] value = message.getBytes();
       metroWriteCharacteristic.setValue(value);
       metroWriteCharacteristic.setWriteType
       (BluetoothGattCharacteristic.WRITE_TYPE_DEFAULT);

       metroBluetoothGatt.writeCharacteristic
       (metroWriteCharacteristic);

   }

}
```

Listing 5. Method to write data to client characteristic.

The method's String parameter is the message to be sent to the client. In order to send data successfully, it first gets the service object from the *BluetoothGatt* by specifying its UUID. From this service object the particular characteristic of interest is obtained by specifying its UUID as well.

4.1.2 MetroDeviceControlActivity

The MetroDeviceControlActivity component is a class that extends an activity class in Android, and therefore it inherits the constants, properties and methods of the activity class. In Android an activity is the component that provides a screen or user interface were users of the application interact, by doing several activities, such as viewing a map, dialing phone and playing mobile games. [12.]

Figure 10 below shows s simple class design for the MetroDeviceControlActivity.

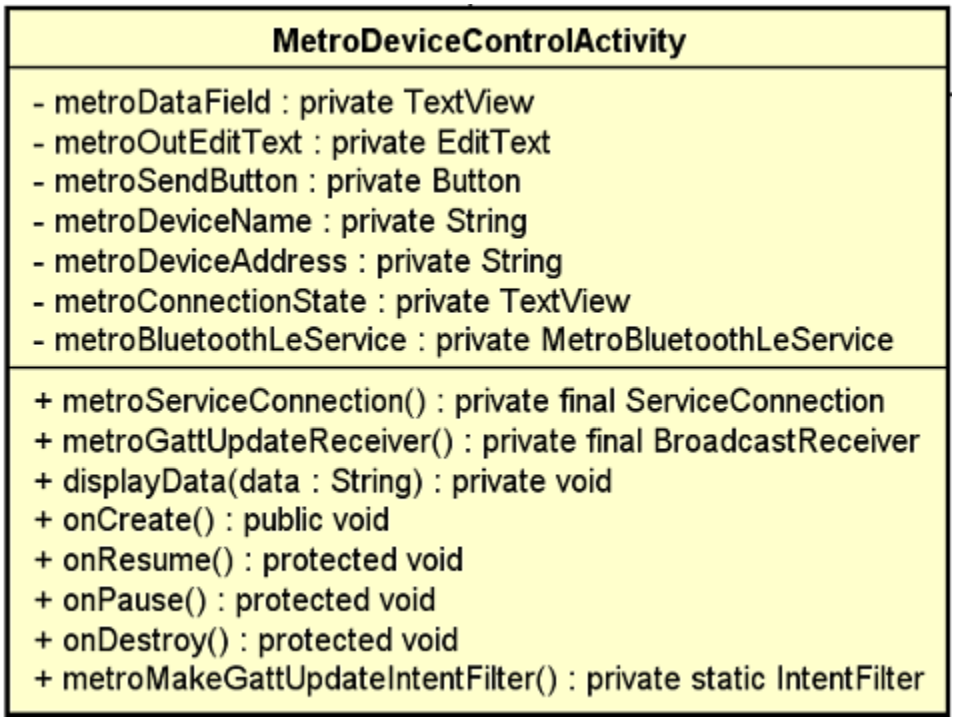

Figure 10. Class design for the activity MetroDeviceControlActivity.

Essential attributes that provide the UI for this activity (MetroDeviceControlActivity) are TextView, an EditText, and a Button. TextView is used for displaying text to a user. EditText is a form of text view (TextView class) configured for text editing. User can input text to the activity from here.[14]. The Button is a class that presents a push-button widget which performs an action when clicked. [15].

The basic roles of the main activity component (MetroDeviceControlActivity) are: extracting the intent for BLE addresses and request for GATT server connection, displaying received BLE data, and sending text data to a BLE peripheral.

In order for the application to connect with the GATT sever of the BLE peripherals, the Bluetooth addresses are required. The addresses are obtained when the application establishes a network connection with the BLE devices discovered during the operations of the *MetroDeviceScanActivity* (or list activity to be discussed in section 4.1.3). The addresses are sent using an intent from the list activity to the main activity, and in the *onCreate* method of the main activity the addresses are extracted into the container list.

```
final Intent intent = getIntent();
list = intent.getStringArrayListExtra("key");
```

Listing 6. Extracting the BLE address into an array list container.

The inter-process communication (provided by the Serviceconnection block) existing between the service and the main activity components aids this GATT server connection, in which the connect method of the service utilizes the addresses contained in the list to establish GATT connections accordingly.

To display received the BLE data the broadcasted BLE characteristic data sent from the service component is received by the *BroadcastReceiver* class instance here in the activity, and the *onReceive* method of this class instance is able to call the *displayData* method to present the received data on the UI for visualization.

```
private void displayData(String data) {
     if (data != null) {
         metroDataField.append(data);
     }
}
```

Listing 7. Method that display the receive data to UI.

The string parameter of the method in listing 7 is the received data to be displayed in the UI, by appending it to the text view object *metroDataField*.

To send text data to a BLE peripheral user needs to edit the text in the text view object *metroOutEditText*. To send an edited text, a user needs to click a button set with an event listener. As soon as the button is clicked the *OnClickListener* code blocks in listing 8 is called to send the edited text to a BLE via the *metrowriteCustomCharacteristic* of the service component.

```
metroDataField = (TextView) findViewById(R.id.data_value);
metroOutEditText = (EditText) findViewById(R.id.edit_text_out);
metroSendButton = (Button) findViewById(R.id.button_send);

View.OnClickListener oclBtnOk = new View.OnClickListener()
{
  @Override
  public void onClick(View v)
  {
   String string = metroOutEditText.getText().toString();
   metroOutEditText.setText("");
   metroBluetoothLeService.writeCustomCharacteristic("Master:"
                                                    + string +
"\n");
  }
};

metroSendButton.setOnClickListener(oclBtnOk);
```

Listing 8. Sending text data to BLE from the MetroDeviceControlActivity.

From listing 8 code blocks the user can view data sent from the BLE peripheral, edit text to be sent to a BLE device, and click a button to send the edited text, all of which takes place in the *oncreate* method of the main activity, so that all the UIs are displayed as soon as the activity comes to the foreground.

Finally in this subsection, these UI resources are visualized on the activity as shown in figure 11.

Figure 11. The application activity showing its UI.

Figure 11 contains text views to display the name, the connection state, the Bluetooth address of the BLE device. It also has the edit text box in which a user can edit text to be sent, a button whose click action sends the text from the edit text box and a text view to display the text received from a BLE server.

4.1.3 MetroDeviceScanActivity

The MetroDeviceScanActivity component of this mobile application extends the android List Activity (also called ListActivity in android programming) class, and by that it inherits its properties, attributes, constants and methods. The list activity is a type of activity in android that binds to a data source, such as an array, a list, and list its items on the screen. [17]. When the mobile application is first launched, the list activity will be the first to appear to the user. It is from here that the application discovers the BLE devices in its vicinity, and list all discovering on the screen for a user to make selections in order to establish a network connection. The main activity is subsequently started with a button click after the user has completed the selection of the devices.

The design class of this component is shown in figure 12 below.

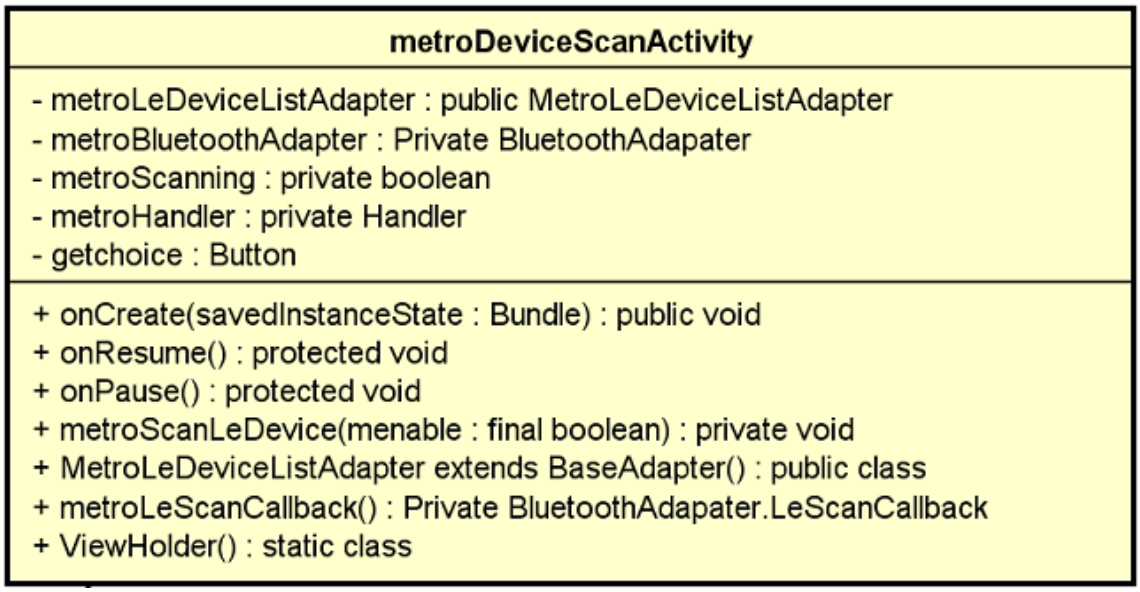

Figure 12. The MetroDeviceScanActivity class design.

The class design of this list activity shows the essential attributes and methods required to perform its role of scanning for BLE devices, displaying discovered devices,

selecting devices to establish connection with, and starting a new activity (MetroDeviceControlActivity). The *MetroLeDeviceListAdapter* attribute is an inner class of this list activity, and it extends a base adapter class (*BaseAdapter* in program syntax). The base adapter class is responsible for storing the discovered BLE devices in a container and binding the elements of this container to the list activity where they are listed in rows for user visualization.

This sub-section mainly focuses on the display of discovered BLE devices along with some UI elements presented as a list on the screen. The MetroLeDeviceListAdapter (base adapter) automatically stores all discovered peripherals into its container, and then its *getView* method customizes both the UI elements (check box, button) and information of the stored peripherals for presentation on the screen. The view object that is finally returned by the *getView* method is customized by declaring a static class to hold the UI element, binding the UI elements to the view object, adding button to the view object, and adding information of the stored BLE devices to the view.

The static class *MetroViewHolder* declaration holds the UI elements that the view object presents to the user.

```
static class MetroViewHolder {
    TextView deviceName;
    TextView deviceAddress;
    CheckBox check;
}
```

Listing 9. Static class to add UI items to the View of the *getView* method.

The *TextView* deviceName displays the name of the Bluetooth device stored in the adapter, while that of the deviceAddress is used to display the device address. *CheckBox* check, provides the means of selecting a device amongst the devices listed on the screen by the view object. To bind UI elements to the view object, the UI resources in the layout directory of the package is added to the view, and its return object assigned respectively to the attributes of the *MetroViewHolder* class.

```
MetroviewHolder.deviceAddress = (TextView)
view.findViewById(R.id.device_address);

MetroviewHolder.deviceName = (TextView)
view.findViewById(R.id.device_name);

MetroviewHolder.check = (CheckBox)
view.findViewById(R.id.check);

MetroviewHolder.check.setOnClickListener(new
View.OnClickListener() {
  @Override
  public void onClick(View v)
  {
    boolean isSelected = ((CheckBox) v).isChecked();
    if (isSelected)
    {
      final BluetoothDevice device =
      metroLeDeviceListAdapter.getDevice(i);
      metroLeDeviceListAdapter.addresses.add(device
                                          .getAddress());
    }
   }

});
```

Listing 10. An aspect of the getView method for adding the UI objects.

Note that the check box has been set with *setOnClickListener* event, which enables it to listen to and get the integer position I, of the view where an item selection was made. The integer position i, corresponds to the index of a stored device, and subsequently extracts the address of this device into the address container of the adapter.

A button element was added to the view also, as shown in listing 11, and its role is to start the main activity when clicked.

```
getChoice = (Button)view.findViewById(R.id.button);

getChoice.setOnClickListener(new  View.OnClickListener()
{
  @Override
  public void onClick(View v) {
        final Intent intent =
        new Intent(MetroDeviceScanActivity.this,
        MetroDeviceControlActivity.class);
        intent.putStringArrayListExtra("key",
        metroLeDeviceListAdapter.addresses);
        if (metroScanning)
        {
           metroBluetoothAdapter
           .stopLeScan(metroLeScanCallback);
            metroScanning = false;
        }
        startActivity(intent);
   }
});
```

Listing 11. Button event in the getView method.

The *setOnclickListener* block does the following when the button is clicked: it embeds the container that is storing the Bluetooth addresses into an intent, and then starts the main activity (*MetroDeviceControlActivity*) with the intent.

The BLE device name and address are obtained from the corresponding index i, of the container and set to the view object, for visualization to users.

```
BluetoothDevice device = metroLeDevices.get(i);
   final String deviceName = device.getName();
   if (deviceName != null && deviceName.length() > 0)
      MetroviewHolder.deviceName.setText(deviceName);
   else
      MetroviewHolder.deviceName.setText(R.string
                                        .unknown_device);
      MetroviewHolder.deviceAddress.setText(device
                                          .getAddress());

   return view;
}
```

Listing 12. Setting the name and address of the BLE device on the view object.

The view object is finally returned to the list activity component for presentation to a user.

In concluding this section, figure 13 below shows the list of Bluetooth devices founded by the mobile app and displayed on the list activity.

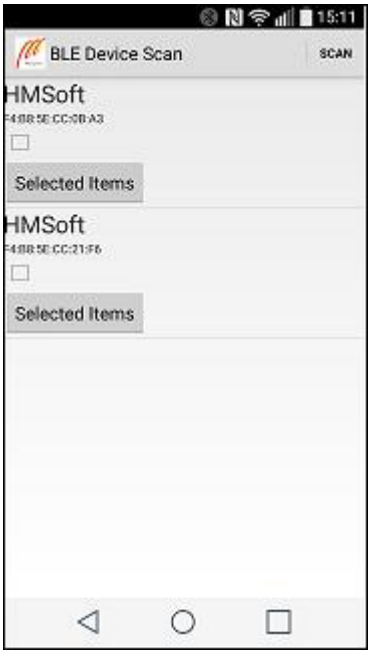

Figure 13. BLE discovered and displayed on the list activity.

As shown in figure 13, a two-number LE device has been discovered by the device and listed in separate rows of the view. Each row contains the name and address of the LE device discovered, a check box to select a device to the established network connection with, and a button to start a new activity when all selections have been made.

4.1.4 The MetroSampleGattAttributes

These is a normal component of the mobile application. It is called normal in the sense that it does not extend any base class of the Android class library, implying that it is neither an activity nor a service we have discussed so far. Figure 14 shows the class design diagram for this component.

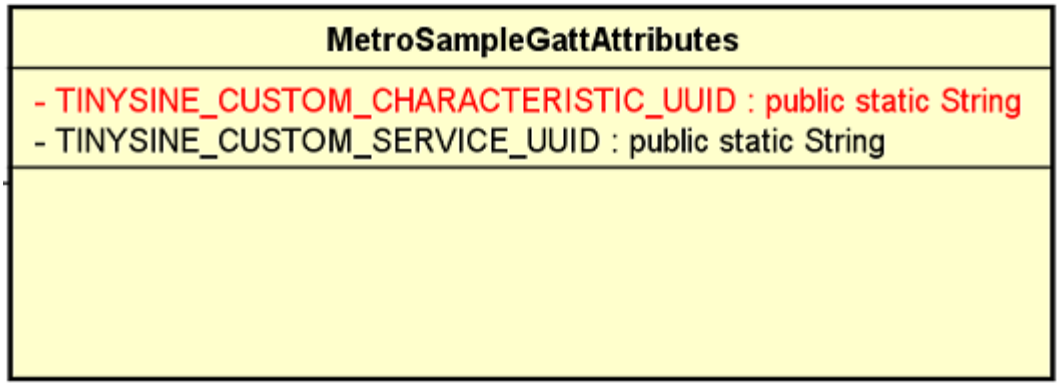

Figure 14. The MetroSampleGattAttributes class diagram.

The purpose of this class is to keep the UUID of any service and characteristic that the application requires to perform some specific operations.

```
public class MetroSampleGattAttributes {
       public static String TINYSINE_CUSTOM_SERVICE_UUID =
                     "0000ffe0-0000-1000-8000-00805f9b34fb";
       public static String TINYSINE_CUSTOM_CHARACTERISTIC_UUID =
                     "0000ffe1-0000-1000-8000-00805f9b34fb";
}
```

Listing 13. The MetroSampleGattAttributes class.

The class contains the string constant TINYSINE_CUSTOM_SERVICE_UUID, used to store the service UUID, and the TINYSINE_CUSTOM_CHARACTERISTIC_UUID for storing the characteristic UUID. Both the characteristic and the service UUIDs used in this project were the values read from the TinySine Serial Bluetooth 4.0 BLE Module when probed with the google app BLE scanner. The UUIDs are useful to access a characteristic when the application wants to write some data to it.

These class does not contain any method, and more UUIDs that might be required by the application, for example sensor's UUIDs, can be added here. In this way the application programming becomes neater, well organized and understandable.

4.2 BLE Peripheral

The BLE peripheral refers to the combination of both the serial Bluetooth 4.0 BLE module and the Arduino UNO device, and the combination is a must in order for the BLE module to be capable of performing its networking role. The BLE is the source of Bluetooth networking connection for this combination, while on the other hand an Arduino device energizes or powers the BLE, processes the data received via BLE and passes this data for example to be viewed in the serial console of a computer.

The TinySine serial Bluetooth 4.0 BLE module can perform both the role of a master and a slave. It can acquire PIO data without other MCU. The specifications are enormous such that;

- size of the data in byte it can send and receive is not limited
- operational frequency is 2.4GHz ISM band
- applies the Gaussian Frequency Shift Keying Modulation
- service UUID for a central is FFE0
- service UUID for peripheral is FFE1
- pairing pin is 000000.

Figure 15 below shows a TinySine Bluetooth 4.0 BLE Module.

Figure 15. A TinySine Bluetooth 4.0 BLE Module. Copied from TynySine (2015) [18].

The AT commands are used to communicate with the internal of the BLE Module, and several of such commands exist, to query and set some internal conditions in the module, for instance;

- To query module MAC address of the device, send AT+ADDR, then it responds with and OK+ADDR:MAC Address.
- To query and set the module PIO output state, after power supplied, first send “AT+BEFC?” to query the module for a parameter ([para]) that needs to be set, and it responds with an “OK+GET:[para]” message. To set the parameter, send the command AT+BEFC[para], values of a parameter ranges from 000 to 3FF. If the changes made with the set command is successful, an OK+SET:[para] is received.

[18.]

An Arduino is an open-source device platform that has simplified its hardware and software usage. It is used for doing the following: control a motor, turn on LED, powering and communicating with a sensor. Depending on the application, Arduino is programmable through its Arduino IDE on the computer by uploading code instructions to the microcontroller on the board. [19.] Varieties of Arduino boards are available in numbers, and in this project the Arduino UNO was used for carrying out implementations. There are 14 digital input/output pins and 6 analog inputs pins in Uno, and figure 16 shows a typical Arduino Uno.

Figure 16. An Arduino Uno. Copied Farnell (2016) [20].

An Arduino Uno is powered with an external power supply, or through a USB connection, and the fourteen digital pins can act as input or output. Pins:3, 5, 6, 10 and 11 provides means for Pulse Width Modulation (PWM). [20.]

Illustrated in figure 17 below is the hardware connection established between the TinySine Bluetooth 4.0 BLE module and the Arduino UNO in order for the combination to perform its needed functionality of establishing a Bluetooth network connection with the mobile application, and to send data through the Bluetooth network to the mobile application as well.

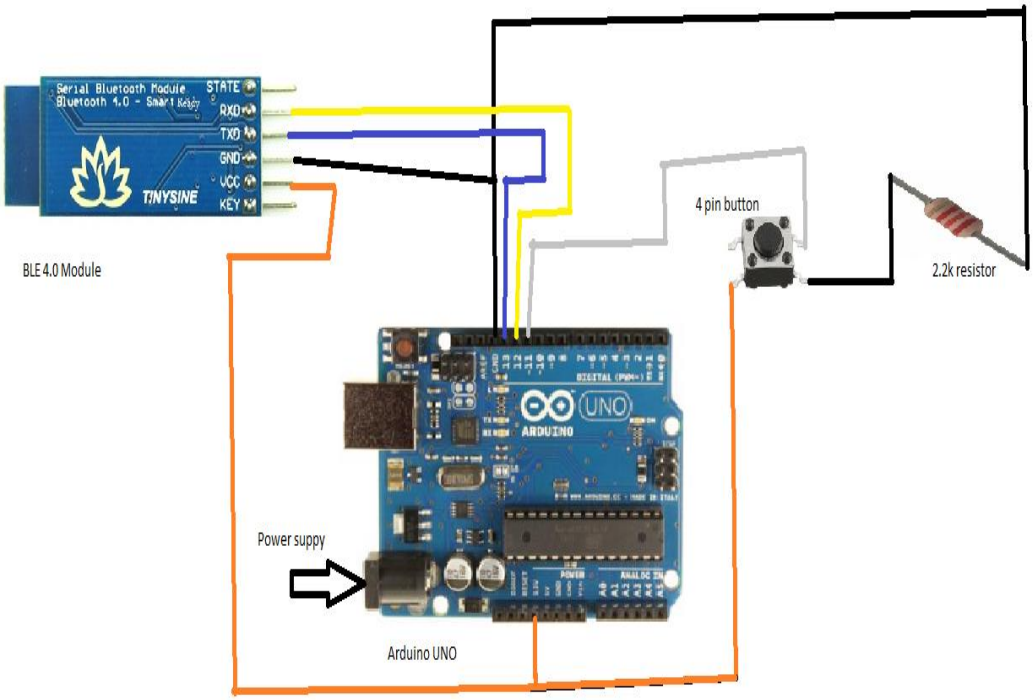

Figure 17. Connection diagram of the BLE peripheral

From figure 17 digital pins 12 and 13 on Arduino Uno are dedicated for receiving and sending data respectively while pin 11 was set aside for the push button, either the USB port or external power pin is the point of powering the device. On the other hand, on the BLE module, pin RXD is used to receive data, while pin TXD is used for sending data. GND is the point of ground connection, and the module shares a common ground with Arduino. Pin VCC is the point of powering the BLE module, and it gets its power supply from the 5v source in the Arduino. The four-pin button is used to create an event for the Arduino platform, so that when it is press, the BLE peripheral sends a text to the mobile application. The input pin of the button was connected to the 5v source of the Arduino, while the output leg was connected to both pin 11 and a 2.2kΩ resistor called a pull-down resistor.

Listings 14, 15, and 16 below describes the stages of the program code uploaded into the arduino, enabling communication or text data to be sent from the BLE peripheral to the mobile application.

The program requires the Serial library to perform the needed role. To ensure this, the SoftwareSerial.h was included. Digital pin 13 is assigned to the transmit (TXD) pin, while the digital pin 12 is assigned to the receive (RXD) pin of the of the BLE module 4.0, and then pins 12 and 13 were enable for Bluetooth operations.

```
#include <SoftwareSerial.h>
int bluetoothTx = 13;
int bluetoothRx = 12;
SoftwareSerial bluetooth(bluetoothTx, bluetoothRx);
int val = 0;
```

Listing 14. Assigning pins.

At the basic setup stage the serial communication port and the Bluetooth port are set to 9600 Baud, and pin 11 set to input mode to be used for the push button.

```
void setup() {
  Serial.begin(9600);
  bluetooth.begin(9600);
  Serial.println("setup complete.");
  pinMode(11, INPUT);
}
```

Listing 15. Setup code block to set basic resources.

The loop block loops continuously, and here the button event takes place.

```
if(bluetooth.available())
{
    char toSend = (char)bluetooth.read();
    Serial.print(toSend);
}

 else
{
     val = digitalRead(11);
     if (val == HIGH) {
     String string_BLE = "Hei Olet BLE 2";
     bluetooth.print(string_BLE + "\n");
 }
```

Listing 16. The loop block

The loop first checks if the Bluetooth communication port is available, and if the test is true, then it receives and print on the console, the data sent from the mobile application through the Bluetooth network. Subsequently, the block reads the input status of pin 11 (button), and when it is high, it implies that the button has been pressed, then string data is sent to the mobile application.

4.3 Demo communication between the mobile application and the BLE peripheral

In this demo the mobile application discovered two BLE peripheral in its vicinity, both devices were selected and a Bluetooth connection was established.

Figure 18 on the left shows the list of the two LE devices found by the mobile application in its vicinity before establishing connection with them.

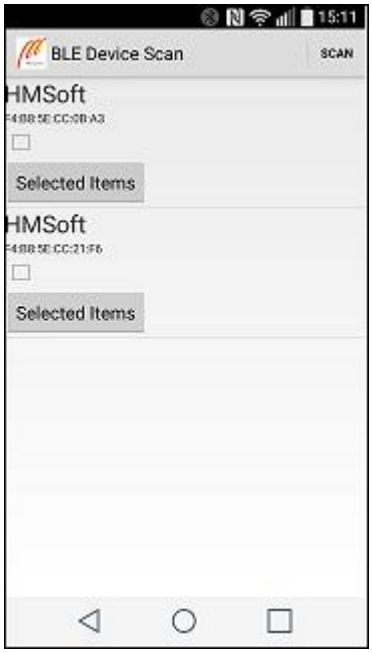

Figure 18. Display of found LE devices

Figure 19. Display of text data

Figure 19 on the right shows the data sent from both devices after establishing connection. BLE device no.1 sent a string data “Hello Am BLE 1”, while BLE device no. 2 sent the data string “hei Olet BLE 2” successfully to the mobile app as clearly shown.

5 Bi-directional Communication and the User Experience

The last chapter is the implementation section, in which an android mobile application was developed to communicate with multiple BLE peripheral in a piconet. The mobile application developed has the ability to receive text data from multiple BLE peripherals in the network. The project implementation has two aspects; developing the mobile app, and programming of the peripheral device an Arduino device which supports the BLE module to perform its networking role of sending and receiving data. The majority of the coding and developmental work was done on the mobile app. This section focuses on two areas of the mobile applications, the bi-directional communication between the mobile app and the BLE peripherals, and the user experience with the list activity user interface.

5.1 Bi-directional communication between the mobile app and the BLE peripherals

The Bluetooth network type that is formed by the mobile application with the BLE peripherals is called a piconet. The number of active devices that make up a piconet ranges from two to eight devices. Usually a single device amongst all of the devices within this network plays the role of the master (central device), while the rest of the devices are usually slaves (peripheral devices). [21.]

Figure 20 below shows an example of the piconet formed by the mobile device and the BLE.

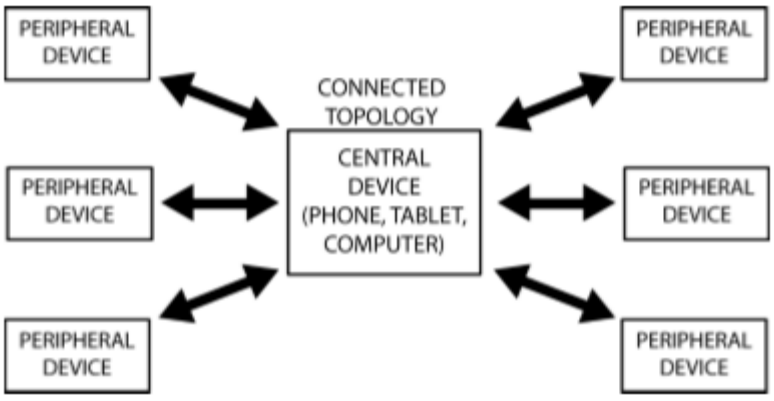

Figure 20. An example of multiple connections of peripherals with a central device in a piconet. Copied from Townsend (2015) [1,25].

As shown in figure 20 the central device can be any of the following; a tablet, a computer and a mobile phone. Regarding this project, our central device is an android mobile phone. The peripheral device is the combination of the BLE and the arduino platform. The arrow in the figure depicts a two-way communication between the central device and the peripherals.

The reason why all peripherals were able to write to the central device, and not the central being able to write all peripherals was traced to the *BluetoothGattCallback* interface class and the write characteristics method in the service component of the mobile app software.

Firstly, let us examine the aspect of reading data from the peripherals by the app, in order to understand why it was able to receive data from all the connected BLE peripherals. Now recall that in the service component of the mobile application is a *BluetoothGattcallback* class instance, which is responsible for implementing a *BluetoothGatt* callbacks. The *BluetoothGatt* provides events that the mobile application cares about from the connected BLE, such as a change in BLE characteristics, a change in characteristic read, and a change in characteristic write. It is the *BluetoothGattcallback* object in the application that listens and implements the appropriate callback methods in response to the event generated by the changes notices from the BLE characteristics.
Below in listing 17 is an aspect of the *BluetoothGattcallback* block from the service component of the mobile app.

```
private final BluetoothGattCallback metroGattCallback =
                                        new BluetoothGattCallback()
{
  @Override
  public void onCharacteristicRead(BluetoothGatt metrogatt,
              BluetoothGattCharacteristic metrocharacteristic,
              int status)
  {
        if (status == BluetoothGatt.GATT_SUCCESS)
        {
            broadcastUpdate(METRO_ACTION_DATA_AVAILABLE,
                            metrocharacteristic);
        }
  }

};
```

Listing 17. The *BluetoothGattCallback* with an aspect of its callback method,

The *onCharacteristicRead* method is the one fired in response to any BLE characteristics sending data to the mobile app in the network. When a characteristic server successfully sends data to the application, the status variable is set true. In that case the particular BLE characteristic which is sending data is passed unto the *broadcastUpdate* method. The *broadcastUpdate* is responsible for extracting the data that is contained in the characteristic, and subsequently broadcast the extracted data throughout the application.

The following behavior has been observed from the operations of the *onCharacteristicRead* callback method;

- At every instance when a BLE characteristic sends a data to the app, the app sees and knows the particular characteristic that is sending data; for example the characteristic from a BLE 1 device is different from the characteristics of a BLE 2 device.
- The peripherals goes ahead to send data to a listening app, without the app requesting the data from the peripherals. So there is no polling taking place as in other systems, but the app only responds to the callback methods of the BluetoothGattCallback object whenever changes occurs on any of the BLE characteristics, and with this capability there is energy saving for the system.

Next we examine why the application is unable to send data to multiple devices on the piconet, excepting only a single component. Two methods that feature when the apps are sending data to the peripherals are; The *onCharacteristicWrite* callback method in the *BluetoothGattCallback* interface class, and the *metrowriteCustomCharacteristic* method of the service component.

```
{

  @Override
  public void onCharacteristicRead(BluetoothGatt metrogatt,
              BluetoothGattCharacteristic metrocharacteristic,
              int metrostatus)
  {
        if (metrostatus == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(METRO_ACTION_DATA_AVAILABLE,
                            metrocharacteristic);
        }
  }



 @Override
 public void onCharacteristicWrite(BluetoothGatt metrogatt,
              BluetoothGattCharacteristic metrocharacteristic,
              int metrostatus)
 {

        if (metrostatus == BluetoothGatt.GATT_SUCCESS) {

        }
 }

};
```

Listing 18. An aspect of the BluetoothGattCallback with its *onCharacteristicRead* and *onCharacteristicWrite* methods.

The difference between the *onCharacteristicRead* and onCharacteristicWrite methods is that the former is called whenever a peripheral in the piconet sends data to the mobile app, while the latter is called when the app sends data to a particular BLE. Note that the different BLE characteristics are able to transact with the app at various instances by sending data through the *onCharacteristicRead* method. On the other hand, the *onCharacteristicWrite* method does not have the ability to select a particular characteristic it is writing to, but all it is capable of doing is to respond to the event generated by the BLE module that the app has just sent data to it, either successfully or otherwise.

The metrowriteCustomCharacteristic method is called when the app basically wants to send data to a BLE module characteristic in the piconet. When this method has finished sending the data, the receiving BLE sends feedback about the condition of the data to the app. This feedback is in the form of an event that the *onCharacteristicWrite* method in listing 18 will respond to.

```
public void metrowriteCustomCharacteristic(final String message) {
            BluetoothGattService metroCustomService =
            metroBluetoothGatt.getService(UUID.fromString(
            SampleGattAttributes.TINYSINE_CUSTOM_SERVICE_UUID));

            BluetoothGattCharacteristic metroWriteCharacteristic =
            metroCustomService.getCharacteristic(
            UUID.fromString(MetroSampleGattAttributes
            .TINYSINE_CUSTOM_CHARACTERISTIC_UUID));

    if (message.length() > 0) {
        byte[] value = message.getBytes();
        metroWriteCharacteristic.setValue(value);
        metroWriteCharacteristic.setWriteType(
        BluetoothGattCharacteristic.WRITE_TYPE_DEFAULT);
        metroBluetoothGatt.writeCharacteristic(
        metroWriteCharacteristic);
    }

}
```

Listing 19. MetroWriteCharacteristic method for sending data to a BLE in the network.

These method has a string argument which is the data it wants to send. Firstly, it gets the service of a BLE by specifying the service UUID. The service UUID of the TinySine serial Bluetooth 4.0 BLE module used in this project is “0000ffe0-0000-1000-8000-00805f9b34fb”, and all the modules used have the same UUID as revealed by the BLE scanner application. The service object from the GATT server when obtained, may contain several characteristics. A get characteristic method is called in order to acquire the characteristic of interest by specifying characteristic UUID. The characteristic UUID for this module is “0000ffe1-0000-1000-8000-00805f9b34fb”.

The method then embeds the string data to be sent into the characteristic, before sending it. The limitations to this method are the following:

- It can only send data to a particular BLE amongst all devices in the network.
- It does not have the ability to select one or more characteristics it is interested in sending data to.
- All BLE in the network have the same characteristic UUID, despite these similarities. The *metrowriteCustomCharacteristic* method that sends the data from the mobile app accepts only a single characteristic as its argument. This method does not have any capability to broadcast data to all BLE in the network; it only sends data directly to a particular characteristic.

- Only a BLE peripheral was receiving data from the app.

A possible suggestion or solution that can enable the mobile application send data to multiple BLEs is by;

- Specifying the handle UUIDs (values) for the characteristics which vary in all BLE, in order to acquire the specific characteristic that is to be written. The reason this could not be achieved at the time of this project is because the android *BluetoothGatt* libraries do not contain functions and classes to perform this kind of operation.
- Creating a multiple *BluetoothGattCallback* for each BLE devices, and binding their characteristic to each of them, and a new thread will be created for each to send and receive data bi-directionally.

Note that it is important for a central to be able to send data to multiple connected peripherals, so that it can set and unset certain features of the GATT server, for example their permissions.

5.2 User experience with the list activity user interface

A very important feature that gives a user an ease of using an application, is the quality of the user interface, where a user interacts with the app. User expectations are that the interfaces should be user-friendly; in that case they should be able to perform certain tasks or activities without any interference on other sets of user interface components that can trigger an action not required.

The first user interface a user will interact with when the mobile application is launched or first started is created with a class called the list activity. On starting the app, the list activity begins scanning for any BLE devices in its vicinity. Any discovered Bluetooth devices are stored in a container of a base adapter class, an inner class of the list activity. The base adapter class binds the stored devices to the list activity class, so that the discovered devices are listed in rows on the screen, where users are able to select the BLE devices to establish a connection with. Figure 21 shows the mobile app list activity interface with the discovered devices listed on the screen.

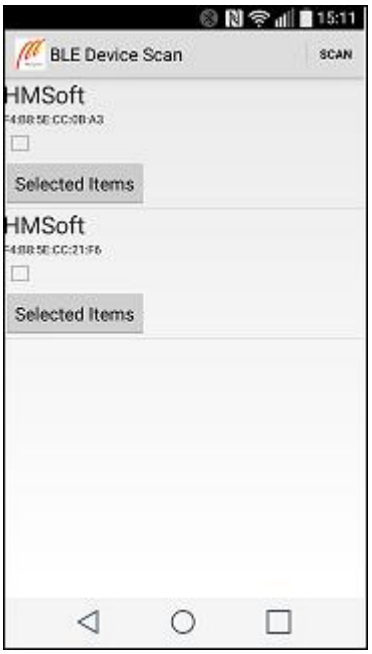

Figure 21. The list activity interface showing listed devices.

In figure 21 the device name, Bluetooth address, a checkbox, and a button have been listed on every row of the list activity screen. There are two devices discovered by the app, as shown in figure 21. The button in every row performs the same function, of starting the main activity, where all data exchange is visualized.

The only improvement that needs to be made, so that a user can enjoy a smooth usage, is to make the list view interface, display only a single button, positioned either above or below all the listed rows, so there is the need to use a different activity class other than the list activity class which is incapable of correcting these limitations. This will improve the quality of usage, so that unnecessary interference with a button, when a user is attempting to make selections of Bluetooth device is minimized.

Despite all of the above mentioned or highlighted limitations, in order to explain the reason why the mobile app is incapable of sending data to multiple BLE in the same piconet, the project was a success because the goal of the task was met. The goal once again was that a multiple BLE device should be able to send text data to a mobile device acting as a central.

6 Conclusion

The purpose of this project was to develop a framework for the Bluetooth Low Energy for educational purposes, by studying and providing a comprehensive theoretical documentation on BLE. In addition to this goal was the development of an Android application capable of establishing communication with BLE peripherals in a Bluetooth network.

It is known now that the BLE device protocol stack, which governs its communication consists of three major layers: the controller layer, the host layer and the application layer. The major layers are further subdivided into sub-layers, with the controller layer consisting of the physical layer and the link layer, the host layer consisting of the logic link control and adaptation protocol (L2CAP), the attribute protocol (ATT), security manager protocol (SMP), the generic attribute profile (GATT) and the generic access profile (GAP). The application layer is the users and BLE interface, which is the layer in which programming is done. BLE comes from different vendors, and for the different devices to be interoperable they must obey the GATT profile.

A communication link between a Bluetooth Android mobile application and multiple BLE peripherals in a Bluetooth network was successfully established, with the peripherals being able to send text data to the central device as required. Communications with BLE are achieved by accessing them through their service and characteristics UUIDs. One very important behavior of BLE is that its GATT server can act at any time either as a client or as a server.

This educational framework has set the stage for further developments. In the future the BLE peripherals in this project which consist of the BLE 4.0 module and the Arduino platform would be replaced by BLE sensors to measure some physical quantities such as temperature and send the measured data to a central device, which can be a mobile phone, a tablet, or a computer. There is however the need to create a more robust system so that the central device would be capable of implementing a bi-directional communication in the network. With this the central could affect certain configurations on the multiple peripherals it is connected to.

References

1. Townsend K, Cufí C, Akiba, and Davidson R. Getting Started with Bluetooth Low Energy. United States Of America; O'Reilly Media Inc; 2014.

2. Galeev M. Taking advantage of Bluetooth LE advertising mode [online]. Asia: UBM Asia Ltd; 22 April 2013.

3. Google Android Developer. Android Studio Overview. USA: Google; 2016. URL: http://developer.android.com/tools/studio/index.html. Accessed 10 January 2016.

4. Google Android Developer. BluetoothGatt. USA: Google; 2016.
URL: http://developer.android.com/tools/studio/index.html. Accessed 10 January 2016.

5. Google Play. BLE Scanner. USA: Google; 2016.
URL:https://play.google.com/store/apps/details?id=com.macdom.ble.blescanner&hl=e.
Accessed 10 January 2016.

6. Google Android Developers. Services. USA: Google; 2016.
URL: http://developer.android.com/guide/components/services.html. Accessed 10 January 2016.

7. Google Android Developers. BluetoothManager. USA: Google; 2016.
URL: http://developer.android.com/reference/android/bluetooth/BluetoothManager.html.
Accessed 10 January 2016.

8. Google Android Developers. BluetoothAdapter. USA: Google; 2016.
URL: http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html.
Accessed 10 January 2016.

9. Google Android Developers. BluetoothDevice. USA: Google; 2016.
URL: http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html.
Accessed 10 January 2016.

10. Google Android Developers. BluetoothGattCallback. USA: Google; 2016. URL:http://developer.android.com/reference/android/bluetooth/BluetoothGattCallback. Accessed 10 January 2016.

11. Google Android Developers. Activity. USA: Google; 2016. URL: http://developer.android.com/guide/components/activities.html#Creating. Accessed 10 January 2016.

12. Google Android Developers. Activity. USA: Google; 2016. URL: http://developer.android.com/reference/android/app/Activity.html. Accessed 10 January 2016.

13. Google Android Developers. TextView. USA: Google; 2016. URL: http://developer.android.com/reference/android/widget/TextView.html. Accessed 10 January 2016.

14. Google Android Developers. Button. USA: Google; 2016. URL: http://developer.android.com/reference/android/widget/Button.html. Accessed 10 January 2016.

15. Google Android Developers. BroadcastReciever. USA: Google; 2016. URL: http://developer.android.com/reference/android/content/BroadcastReceiver.html. Accessed 10 January 2016.

16. Google Android Developers. ListActivity. USA: Google; 2016. URL: http://developer.android.com/reference/android/app/ListActivity.html. Accessed 10 January 2016.

17. TinySine. TinySine Bluetooth 4.0 BLE module User Manual. China: TinySine Electronics; 2015. URL:http://www.tinyosshop.com. Accessed 10 January 2016.

18. Arduino. What is Arduino. USA: Arduino; 2016.
URL:https://www.arduino.cc/en/Guide/Introduction. Accessed 10 January 2016.

19. Farnell. Arduino Uno. UK: Farnell; 2016. URL:
http://www.farnell.com/datasheets/1682209.pdf. Accessed 10 January 2016.

20.TechTerms. Piconet. USA: TechTerms; May 28, 2008.URL:
http://techterms.com/definition/piconet. Accessed 10 January 2016.

Title of the Appendix

Content of the appendix is placed here.

Title of the Appendix

Content of the appendix is placed here.