Bluetooth Low Energy Application in home automation

Based on Raspberry Pi 3B board

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Electrical and Automation Engineering

Autumn,2019

Xin Zhong

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Automation and Electrical Engineering
Valkeakoski, HAMK

Author Xin Zhong **Year** 2019

Subject Bluetooth Low Energy application in Home Automation

Supervisor(s) Juhani Henttonen, Raine Lehto

ABSTRACT

Nowadays, the electrical devices for home automation are required to be multi-functional, with high design, and user-friendly. To meet the requirement of simple appearance and various functions, many manufactures create end to end(E2E) solutions e.g. for controlling an air purifier on a mobile device as was done for the commissioner here, remote control of devices with less or even no physical switches.

This trend has excited the release of many communications technologies of Machine to Machine(M2M), like Bluetooth, Zigbee, Zigwave, Enocean.

The author was commissioned by a Finnish air purification company, to develop a local wireless control of a stand-alone purifier device. This device would support cloud message control from the commissioner as application end (later referred to as App). Considering the geographic distance for maintenance, initial Bluetooth setup would help to provide the user access to further manual control or a WIFI configuration.

The thesis project focused on building a Bluetooth communication application on responder side, the purifier itself, to exchange data with the initiator device, the commissioners mobile end, which has App for this. The target here was to present a prototype with full functions, a stable connection and discuss about the further implementation on security pairing.

Keywords Bluetooth, air purifier, peripheral, central, encryption

Pages 35 pages including appendices 0 pages

CONTENTS

1 INTRODUCTION

Today air pollution is a serious environmental issue, especially in some industrialized countries which have a high energy demand and modern traffic system. Particles and volatile organic component (VOC) in the air will enter the bodies of all creatures and bring a potential risk of respiratory disease to human beings.

In China, the winter and early spring are the seasons with the highest particulate matter with diameter 2.5 micrometres (PM)2.5 level, also with the highest incidence of respiratory diseases among normal citizens. To protect people from air pollution, stand-alone air purifiers are promoted widely as a part of home automation.

1.1 Background of Commissioner

The Commissioning company (later referred to as commissioner) is a medium- size company in Finland, specializing in clean air technology. Since 2018, the commissioner started the development work of new air purifiers. So far, the air purifier protype was finished, and some devices have already been delivered to customers.

The author of this thesis was recruited by the company to develop a local control solution for the purifiers. To simplify the assembly and set-up work for the end-customers, the device should have built-in Bluetooth available for the initial setup.

Referring to customer feedback on the first-generation air purifier utilizing a classic Bluetooth protocol, the time delay from tablet starting to transmit the command to the purifier executing the command was intolerably long. It did not accept the transfer of long messages. Such issue finally limited the functionality of products. Another problem was that iOS devices do not support a classic Bluetooth protocol. Through discussions was Research and development team (later referred to as RD), it was decided to use Bluetooth low energy (later referred to as BLE) technology as the final solution for local control.

This project was based on an offline Bluetooth control utilized by a stand-alone air purifier.

1.2 Objectives and research questions

The goal was to develop a Bluetooth control system which could control fan speed of an air purifier to eliminate particles, configure the device name and the WIFI settings. This thesis starts from the device’s functionality demands, the leading to the BLE protocol,

and illustrate the applications construction and debugging process. This thesis ends with conclusion of the project results and a further exploration of other flexible and relatively secure pairing method.

The following research questions were raised for the development work:

1. Which Bluetooth technology is the most suitable to our project, classic Bluetooth or Bluetooth low energy?
2. Which hardware platform could meet the function demands?
3. How to integrate the Bluetooth application into the current controller with a message queue connected to the cloud client?
4. Which pairing method to be utilized as a security feature in the next design phase.

1.3 **Hardware introduction of air purifier**

As a stand-alone air purifier, the device examined here combines electrical and mechanical filtering technologies, by pumping the air through an ESD charging part to charge the particles in the air, collecting all particles by multiple filter layers.

The control part of the purifier consists of two boards: the power control board and the program controller board. The power board regulates the power for the high voltage charging module, the fan motor's control voltage, and the program controller which was a Raspberry Pi 3B board as seen in Figure 1.

Figure 1. Raspberry Pi 3B (Raspberry Pi 3 Model B, n.d.)

Raspberry pi is a small single-board computer board, powered by 5V. It is commonly used by many developers as a platform for building applications.
In this project, Raspberry pi 3B was chosen because it had on-board wireless LAN and BLE technology, which were necessary for the wireless control solution.

Considering the start-up scale of the project, the company selected Raspberry Pi and its operating system NooBS because Raspberry had enough number of GPIO pins for

communicating with power board, and flexibility to support variable applications no matter in Python, C, bash. The low cost of Raspberry pi is was also considered as a plus for the development project.

1.4 **Programming language and source library**

In this project, the programming language the author used for constructing, and for debugging test with Python.
Python is a high-level programming language, friendly to beginner-level programmers and engineers. It has the featuring:

1. It is interactive.
2. It is Object-oriented.
3. It is interpreted.

When this project was launched, to consider the author's ability and long-term training plan, the company decided to choose Python as the programming language.

The library code used in this BLE project was Pybleno, which is the Python version of the Bleno library on Github. It is an open source code library written by Adam Langley. A direct port of the Bleno Bluetooth LE peripheral role library to Python2/3. The Pybleno library logic was from Sanddep Mistry, the designer of Bleno, which was built with JavaScript. (Langley, 2017)

The original version of Pybleno does not finish the encryption files in its host controller interface (later referred to as HCI). The author implemented the encryption files in the source code to achieve the basic pairing process. The implemented files were not uploaded to contribute to the original library for the confidential reasons. In following chapters, the implementation of encryption for BLE pairing will be presented as part of the project of this thesis.

1.5 **Debugging tools**

During the development work of the project, the author used a public Bluetooth application to test the related function status and the pairing process. There are several applications available in the market, and the most common options are Light Blue® and BLE Scanner. Light Blue® and BLE Scanner are available in Google Play and iOS store. They could search the Bluetooth devices nearby and allow users to inspect their peripherals. The app allows user to browse characteristics and services, and to write value in hex or text string.

2 **EXPLORATION OF SOLUTION**

Starting from the development work on the first generation product, the company decided that the program should be able to change the Raspberry GPIO status which was read by the power board to change the level of fan speed control(0-24V) and Hight voltage charging unit(0V or 15V).

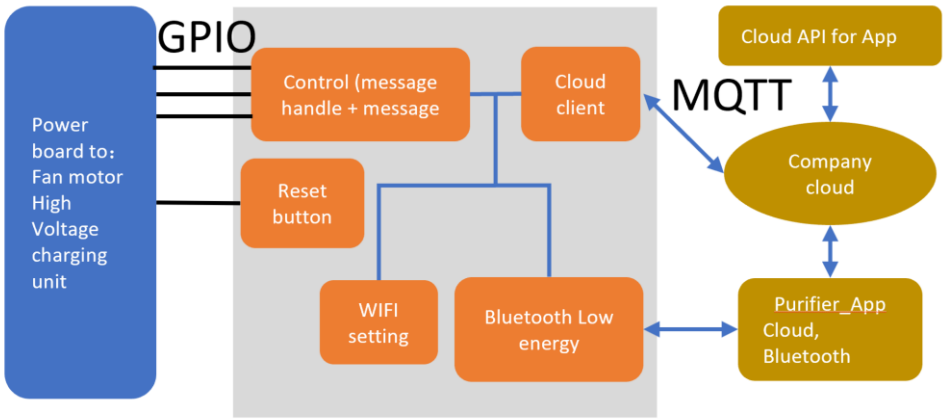

Figure 2. IoT structure of air purifier

Figure 2 shows an overview to the purifier service. The local control part is the RPI module on the left, with a LED panel as a model. The controller block can change the fan speed from 0 to 5 and enable or disable the charging mode based on the commands from: cloud, the Bluetooth or timer.

2.1 **General requirement**

To make the style according to Scandinavian design, the original product appearance design lost the physical switch. It required wireless control of user configuration, machine status, control message between App on commissioner's hand and purifier.

The commissioner is building an automatic control mode to flexibly change the purification level based on the local air quality. It requires individual cloud space to calculate the air quality data and send out the related commands.

The production is set in Finland, the distribution and selling business are global. It required the product design to be easily assembled, without extra work on local network setting. It also means the user was able to set the initial WIFI configuration without any display or keyboard. In this situation, Bluetooth was the best option, as the programme could be built inside Raspberry in advance. Also, Bluetooth requires no extra gateway or hub to compile.

2.2 Requirements of Bluetooth functions

Based on BLE technology, the purifier was required to do the following functions:

1. To send the control messages for different fan speeds, to switch between the charging mode and the automatic mode.
2. To read the status messages of item 1, status of cover close, reset button status, power on status, series number, cloud and WIFI status.
3. To write the string messages for the WIFI registration and user configuration
4. To build the security methods for encrypting the pairing process and decrypting the string message received.

2.3 Message queue connection

Message exchange was essential in this project. Each application block was to send or fetch control messages in the control channel and share public messages in the status channel. As in Figure 3, a message queue had already been built before, connecting the cloud client to the local controller. There were different control channels in Redis server to separate the control message source due to the demand of switching the control mode in the previous development phase.

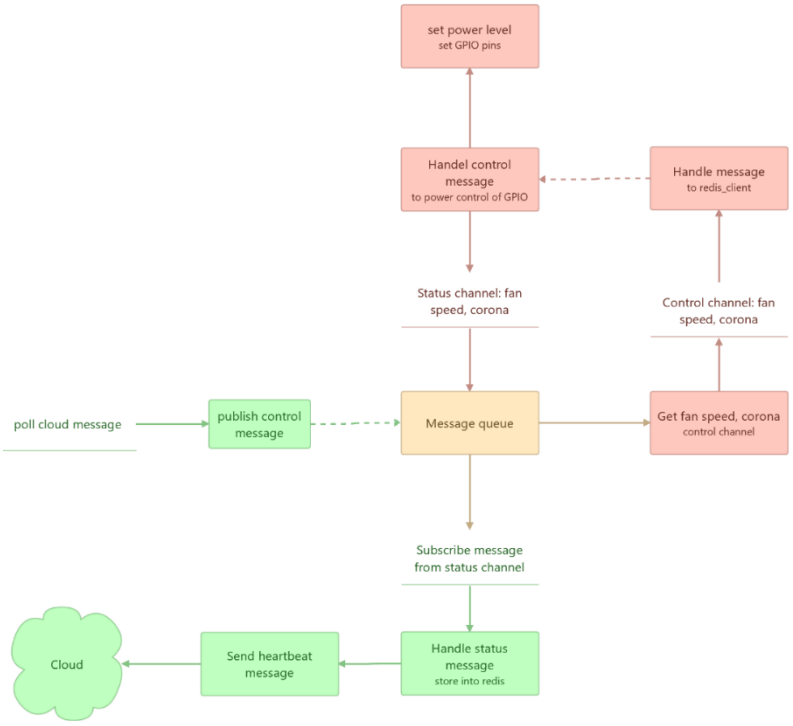

Figure 3. Message transfer between cloud and local control

Message queue connected the cloud client, the main controller as well as the device health reporter to subscribe control command and to publish status message (pub-sub per second. The Bluetooth server, external button controller and timer mode controller randomly published or subscribed depending on the different user cases.

Figure 4 illustrates the procedures that how Bluetooth block sent the control messages to message queue and fetched the status messages.

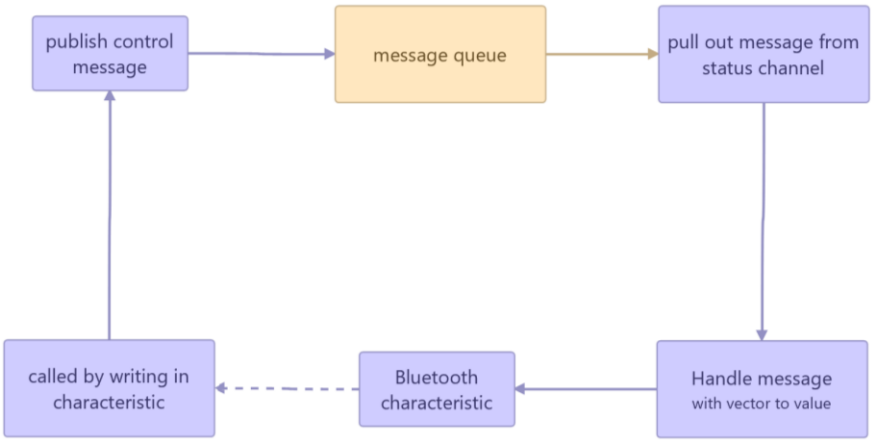

Figure 4. Bluetooth message transfer

3 THEORETICAL SUPPORT TO WIRELESS CONTROL

3.1 Security and protection tool

3.1.1 32-bit Cyclic Redundancy Check (CRC)

CRC is an error-detecting method, commonly used in the digital networks such as I2C, UART, Bluetooth. It could efficiently check the accidental changes in the raw data. Each time the transmitter sends a series of data, it could compute CRC value of message and add it into data. On another side, receiver could calculate the CRC value and check whether computed result matches the CRC value given in data. The calculation is based on the remainder of polynomial division of the message content. The most common CRC method is 32-bit, which has 33-bit divisor length. Its pattern is Formula 1:

$$Polynomial\ pattern = x^{32} + x^{26} + x^{23} \quad (1)$$

$$+x^{22} + x^{16} + x^{12}$$
$$+x^{11} + x^{10} + x^{8}$$
$$+x^{7} + x^{5} + x^{4}$$
$$+x^{2} + x^{1} + 1$$

When algorithm starts, it acts on the bites based on divisor, and the result for each step is the bitwise XOR of divisor with the bites from the first bit 1 on the left side. Then the divisor is shifted to the right for one bit and repeat the previous action until it arrives the right end. The divisor length decides the CRC value length in bits, which means the calculation result could have collision chance with other possibly wrong content. The chance of collision is up to the length of CRC result. 32-bit CRC value length could provide over 4 billion available hash values. It is accurate enough for data verification in normal usage.

3.1.2 Rivest–Shamir–Adleman crypto system (RSA)

RSA is a public-key cryptosystem, widely used for data encryption during transmission. This cryptosystem creates public key and private key. The user will store the public key and share the private key to another side. The public key is used to encrypt message, and private key is used to decrypt message.
The RSA algorithm process has four steps:

- Generate key
- Distribute key
- Encryption
- Decryption

The basic principle is as Formula 2:

$$(m^e)^d \equiv m\ (mod\ n) \qquad (2)$$

The index e, d and n are large positive integers. It is required to make $(m^e)^d$ is the modular exponentiation for all integers m, with modulus n.
It will difficult to find even a hacker already knows the e, d and m. Even if the e and d exchange the position, the relation among m, e, d, n is still the same. In this formula, d is the private key exponent, and n is released as public key. The m is the message. Firstly, the message m was transformed into integers m, which should satisfy: $0 \leq m < n$

Then the message was encrypted as Formula 3:

$$C \equiv m^e (mod\ n) \qquad (3)$$

When the encrypted content was received by another side, the message should be decrypted with Formula 4:

$$C^d \equiv (m^e)^{\mathrm{d}} \equiv m\ (mod\ n) \qquad (4)$$

Then the m is the original message after calculation.

3.1.3 Wi-Fi Protected Access (WPA)

WIFI protected access is a security protocol and security certification program developed by WIFI Alliance for the security of wireless network among computers. In common embedded system, there is a supplicant support for WPA, called WPA supplicant. To apply WPA supplicant in application layer, there is a text-based program as frontend for interacting with it, called WPA_CLI. In embedded system, it supports command to display the current status, to manage the configuration and to request new input.
To use WPA_CLI, the control interface should be configured under normal user account.

3.1.4 Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker (Introduction to Redis, n.d.). Compared to other data structure stores which are on disk, Redis is in-memory, making the response speed much higher. That is the reason for the general purifier project select Redis as database application. It does not require table, could store strings, lists, sets, hashes and sorted set, meeting the demand in this project to publish and subscribe various parameters between cloud client and local device.

3.2 **Bluetooth low energy**

3.2.1 Introduction of Bluetooth Low Energy

Bluetooth Low Energy (BLE) technology appears in market with the release of Bluetooth 4.0 in 2011. compared to the classic Bluetooth technology, it consumes less power. As it has high data transfer rate up to 1Mb/s, the actual connection time are only few milliseconds. BLE operates in the 2.4 GHz, same as the classic Bluetooth. Compared with the classic Bluetooth, BLE is more suitable for devices with less data demand, for example: the blood pressure monitors, the industrial monitoring sensors.

BLE separates data into packets before transmitting it. The valid transfer range is within 77m. It is normally recommended to have maximum 8 responder devices in BLE protocol. Its latency is only 6ms compared to 100ms in classic Bluetooth protocol. The company decided to have BLE as short-range communication solution. It is because the classic Bluetooth in the previous application showed a long latency during the connection in each message transfer. BLE could provide a more stable and longer

connection, raising the speed of data writing and reading, generally optimizing the user experience.

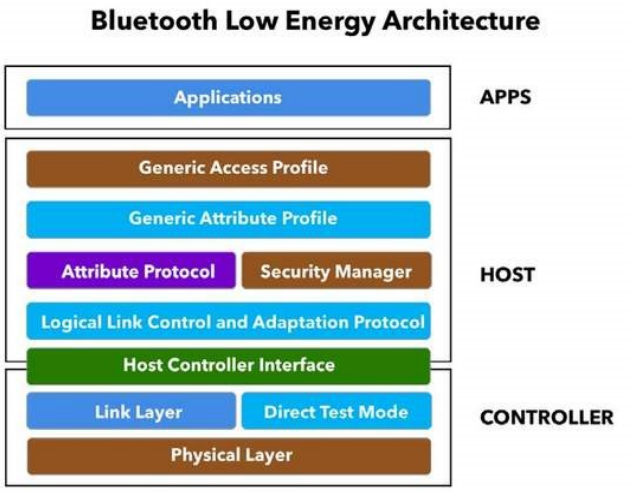

Figure 5. Architecture of BLE (Afaneh, n.d.)

The Figure 5 shows the architecture of BLE. The application was on the top layer, it was supported by the generic access profile, and the generic attribute profile. Attribute is on GATT client side to allow the device to discover the GATT server service. Security Manager is on both sides to define the pairing and encryption methods.

Below the Logical Link Control and Adaptation Protocol is the Host Controller Interface (HCI)layer. It standardizes the communication of host layer and controller layer.

First, the advertiser will be advertising its universal unique identity code and its pre-defined name on the application layer. When responder received the signals, it will initialize the connection by sending an connect request and its role changed as an initiator on the connection layer. After the connection is built, the initiator will become the initiator on link layer. Then the advertiser will become responder. Finally, the initiator device got access to do read or write actions to the responder device.

3.2.2 BLE GAP, GATT, ATT

BLE profiles defined the process how the BLE devices connect and communicate with each other. BLE Generic Attribute Profile (later referred to as GATT) defined a data table to manage the fields of devices and their status, operations. It is a clean and hierarchical structure consisting of Services, Characteristics, Descriptors, shown in Figure 6.

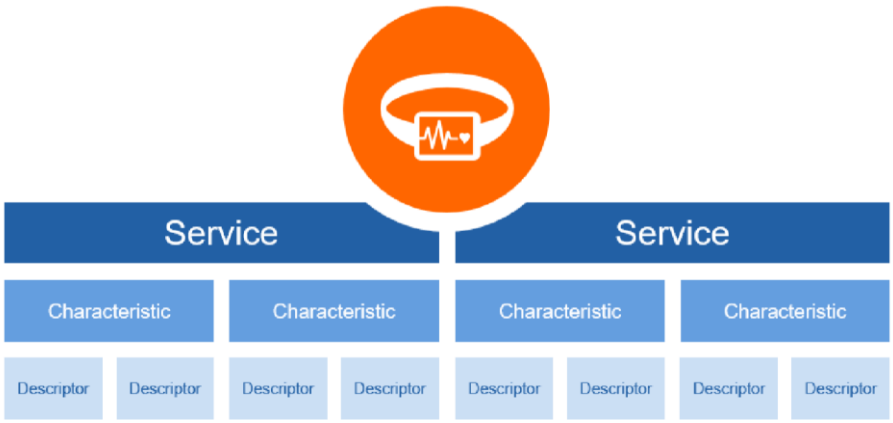

Figure 6. Services, Characteristics and Descriptors (Developer Study Guide: An introduction to Bluetooth Low Energy Development , 2018)

Services are the top-level containers, having multiple characteristics and some primary feature of devices. Characteristics are the independent items with defined data type, value, and some unique operations like security, notification. The read and write access could be decided by developers.
Descriptors normally transfer text or some meaning of operations like notifications.

In BLE development, the devices to host a set of services which containing characteristics, and descriptors are called GATT server. Devices which accessed services, read value of characteristics and descriptors are called GATT client. BLE also defined the connection between GATT server and client, as Generic Access Profile (later referred to as GAP). GAP defined the roles of BLE devices: peripheral, Central, Broadcaster, Observer, and define their advertising process and security feature:

- Peripheral devices always start advertising, inviting connections, always sensor or a device
- Central devices scan a peripheral device advertising and decide to connect it, normally a mobile device or pc.
- Broadcaster advertises but does not accept connections.
- Observer scan the advertising packets from broadcaster but does not connect it.

Broadcaster and observer are usually in Beacon application. Attribute Protocol (later referred to as ATT) is the communication layer of BLE stack to allow GATT client transfer data with GATT server after connection is done. (Developer Study Guide: An introduction to Bluetooth Low Energy Development , 2018)

3.2.3 BLE package format

BLE has 40 physical channels in the 2.4GHz brand and each channel shares 2MHz, shown as in Figure 7. It defines advertising and data transmission types. 3 channels are for

advertising (channel 37, 38, 39) and other 37 channels are used to transfer data. The selection of these 3 channels are for avoiding the interference of WIFI, because 2402MHz, 2426MHz and 2480MHz are not commonly used by WIFI.

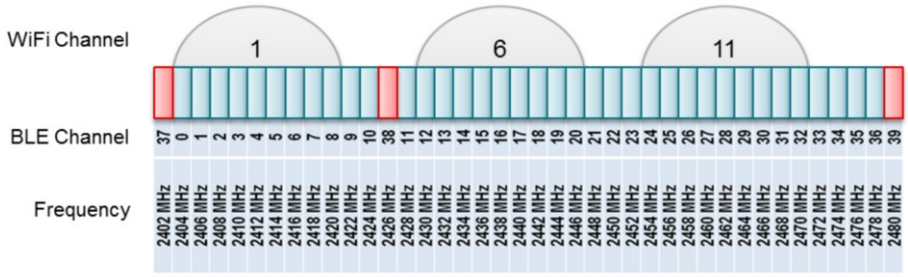

Figure 7. Frequency Band and Channels, Bluetooth low energy (Lindh, 2016)

Advertising in BLE is to broadcast the device name and address of peripheral to the central.

Shown as Figure 8 and 9, the package format consists of four components: preamble (1 octet in Blue4 and 2 octets in Bluez 5), access address (4 octets), Protocol Data Unit (2 to 257 octets) and Cyclic Redundancy Check (3 octets).

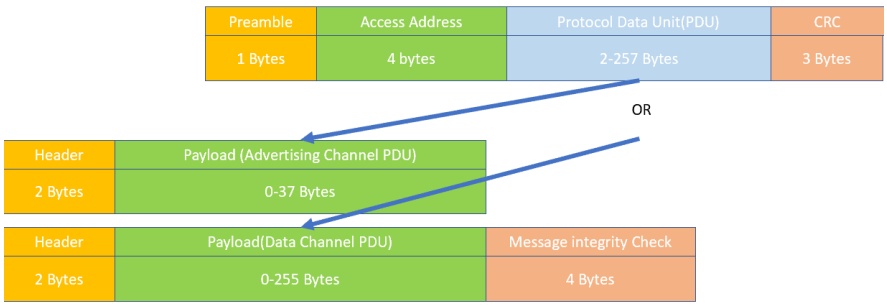

Figure 8. BLE package format. Adapted from Microchip 2019 (Microchip Technology, 2019)

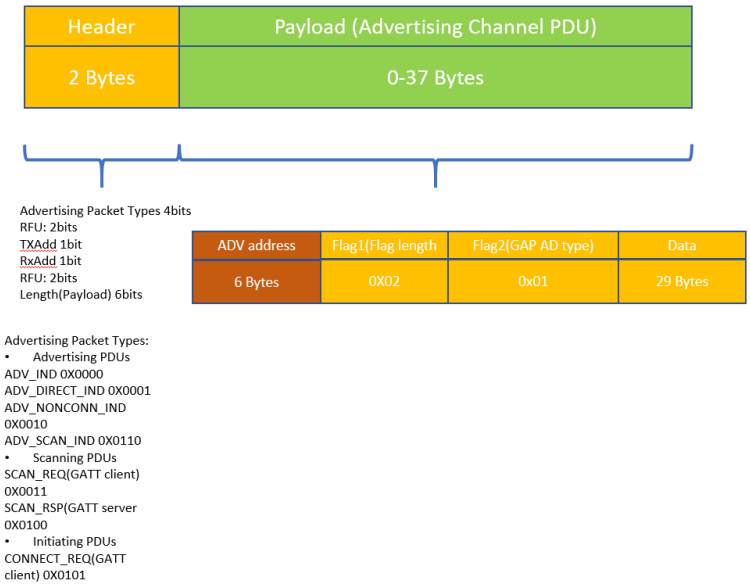

Figure 9. Protocol data unit. Adapted from Microchip 2019 (Microchip Technology, 2019)

In this format, PDU is important because it could decide the packet type whether data or advertising. The PDU consists of header and payload. Payload contains the information the packet delivery, and header contains the PDU type, transceiver and receive address and length.

3.2.4 Bluetooth pairing process

Bluetooth pairing and encryption process is done by Security Manager Protocol (later referred to as SMP). It defines, receives and transfers pairing command, meanwhile doing encryption by distributing keys. The whole pairing process consists of Pairing and Bonding, to establish keys for encrypting link and then distribute them to share between peripheral and central devices. The current typical pairing process in Bluetooth 4.2 has 3 phases:

- Phase 1: Pairing Feature Exchange
- Phase 2: LE legacy pairing (Short Term Key Generation)
- Phase 2: LE secure connections (Long Term Key Generation)
- Phase 3: Transport Specific Key Distribution

The Figure 10 below shows the BLE legacy pairing in Just Works mode.

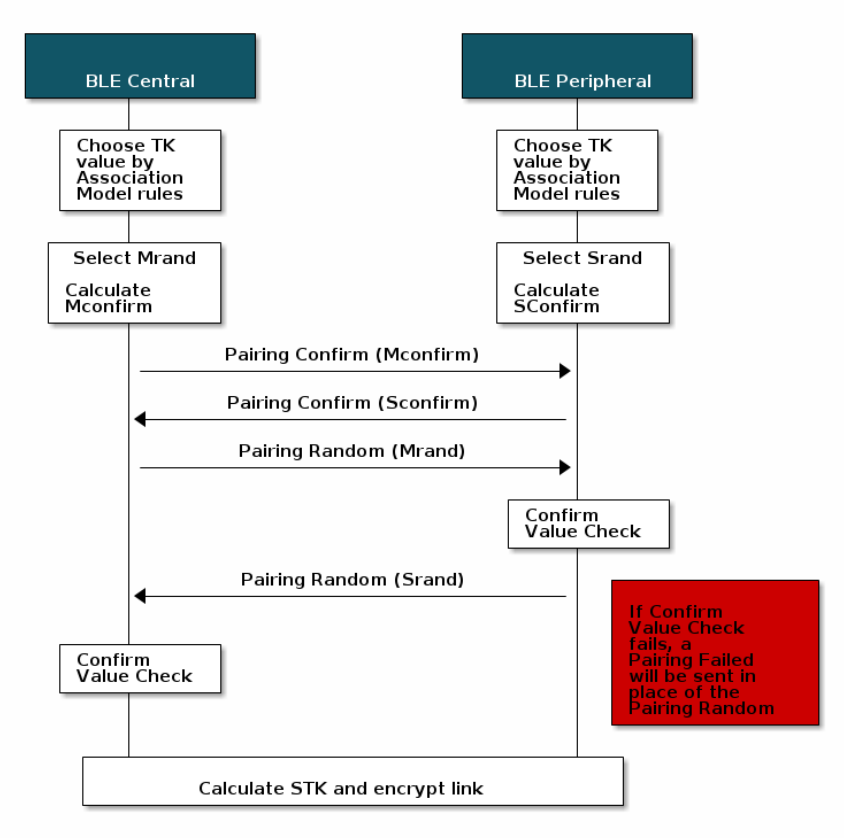

Figure 10. BLE Legacy Pairing process
(Bluetooth low energy Security Fundamentals, n.d.)

3.2.5 Exchange of pairing feature

According to the protocol, Peripheral and client devices need to know input or output features of each other, boding demand, security level and so on. Then the following pairing method could be selected.
First, client device needs to send pairing request to peripheral device. When peripheral device receives It, it will answer client device with a byte array of pairing features shown as in Table 1.

Each time the devices sent message to each other; the message head was an operation code to indicate the action that the sender intended to do. The definition of each operation code is shown in Table 2.

Table 1. Pairing features exchange

Field	Definition	Bits
Operation code		8

IO cap		8
Out of Band (later referred to as OOB)		8
Authentication Request	BF	2
	MITM	1
	SC	1
	KP	1
	Reserved	3
Maximum Encryption Key Size		8
Initiator Encryption Distribution		8

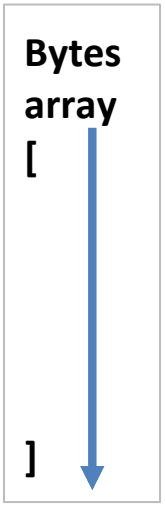

Table 2. Operation code

Operation Code	Description
0x01	Pairing Request
0x02	Pairing Response
0x03	Pairing Confirm
0x04	Pairing Random
0x05	Pairing Failed
0x06	Encyrpt_INFO
0x07	Mater_IDENT

The method of pairing between peripheral device and central device is decided by their own IO capabilities, shown in Table 3. If a peripheral device and a central device both have screen to display, the pairing with pin code or random number is possible. If one of them has no screens or keyboards, the pairing between two devices will be only “Just Works”, which only notifies user to press “Yes” or “No” on the central device.

Table 3. IO Capabilities

IO Cap	Description
0x00	Display Only
0x01	Display Yes No
0x02	Keyboard Only
0x03	NoInputNoOutput
0x04	KeyboardDisplay
0x05-0XFF	Reserved

3.2.6 Key generation

Firstly, central device receives the pairing request, it will generate confirm number array which start with Pairing Confirm command and send it to peripheral device. When peripheral device receives the pairing confirm message, it will remember the confirm number and create its own confirm message starting with Pairing Confirm command and send it back to central device.

Once central device receives the pairing confirm message from peripheral, the device will send the pairing random message starting with Pairing Random command. The message includes the random values list which participated in confirm number generation. As “Specification of the Bluetooth System” document defines the algorithm on both sides to generate the confirm number by random value, both peripheral and central device should have same process for verifying the calculation result.

After that, peripheral device received the random values list, it will generate the expected confirm numbers list and compare it with confirm number received from central devices. If it is correct, the peripheral device will send the value back to central device for random values checking. The mismatch of confirm numbers will cause failure and both devices will receive Pairing Failed command from each other to end the pairing process.

Finally, if confirm numbers on both devices could match the calculation result, the link will be encrypted by short term key (later referred to as STK). Long term key (later referred to as LTK) will be distributed and stored for device bonding. The bonding information will be stored in flash.

4 APPLICATION

This pair purifier control application was based on a BLE technology application. The programming for this project was conducted by Python, and the BLE library utilized was Pybleno, an open source library on Git-hub.

4.1 **Hierarchy structure of peripheral**

As GAP defines, the purifier controller Raspberry Pi 3B should play the role of a peripheral to advertise its Universal Unique Identifier (UUID) and the pre-defined name.

it was designed to use 128-digit UUID to prevent any device around interfering with it if the devices nearby had the same UUID in 16 digits.

To control the device, the items listed in Table 4 and 5 are the individual characteristics with different accesses during the data transfer.

Table 4. Characteristic access for different functions, X showing the features.

Name	Value	Read	Write	Notify
Automatic mode	01, 00	X	X	
Charger	01, 00	X	X	
Cover open	01, 00	X		X
Fan speed	level index	X	X	X
Reset Press	01, 00	X		
Power on	01, 00	X	X	
Serial Number	Hardware ID	X		
Cloud_ Wifi_Connection	Logical status	X		X
Cloud settings_status	01, 00	X		
BTPassword	Hex number		X	

The fan speed had been predefined as the index level as designed in the power controller circuit. A series number was stored in factory configuration file and this would not never change because it was used for recognizing one purifier device during the initial setup. The Cloud-Wi-Fi-Connection Characteristic was to feedback the connection status of WIFI and cloud to app on the central device.

There are other three characteristics in the control service, designed for the long string transfer. Since the maximum ATT payloads was 22 bytes in Bluetooth 4.0 and 4.1, the long string transfer was required to transfer 20 bytes every time to be compatible to different Bluetooth protocol and to reduce the difficulty of CRC value check.

Table 5. Long string characteristics access

Name	**Value**	**Read**	**Write**	**Notify**
String type	DEC	X	X	
String stream switch	BOOL	X	X	
String write	STRING	X	X	

To add extra protection, a fixed password was required to write to get the access to read or write some characteristics. The features for each characteristic in Table 5 were required to be added in the real program.

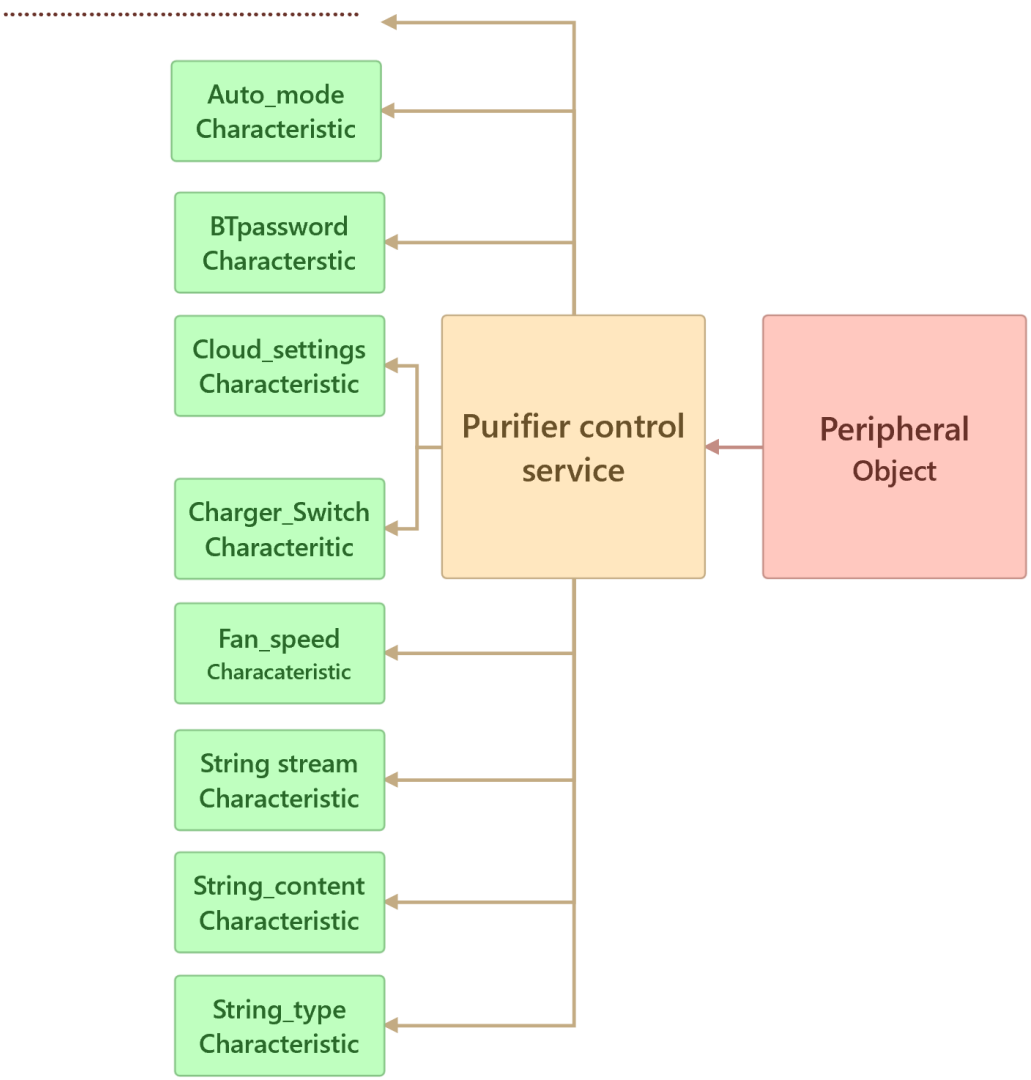

Figure 11. Structure of Peripheral

As Figure 11 shows, the function characteristics were all included in Control service which was a primary service in the peripheral. Taking “Fan-Speed-Characteristic” as example, its properties were defined as “read”, “write”, “notify”. It had its own UUID and received the reading and writing request from the central device which had built the connection with the peripheral already.

Shown as Figure 12, on the one hand, “Fan-Speed-Characteristic” could connect to “Purifier” application object (later referred to as “Purifier”). The reading function would directly call the function from the “Purifier” which packaged the values into control message in a json format and published it into message queue. The control message would be received and executed by the controller object which output status message.

On the other hand, after time delay in pre-defined seconds, the “Purifier” object subscribed to message queue and pulled out of the status message from its status channel. Because there was a vector variable already pointing the specific key, such as “fan speed”, “auto-mode”, the message would be picked out by vector and emited back to one characteristic as notification result or response to the reading request.

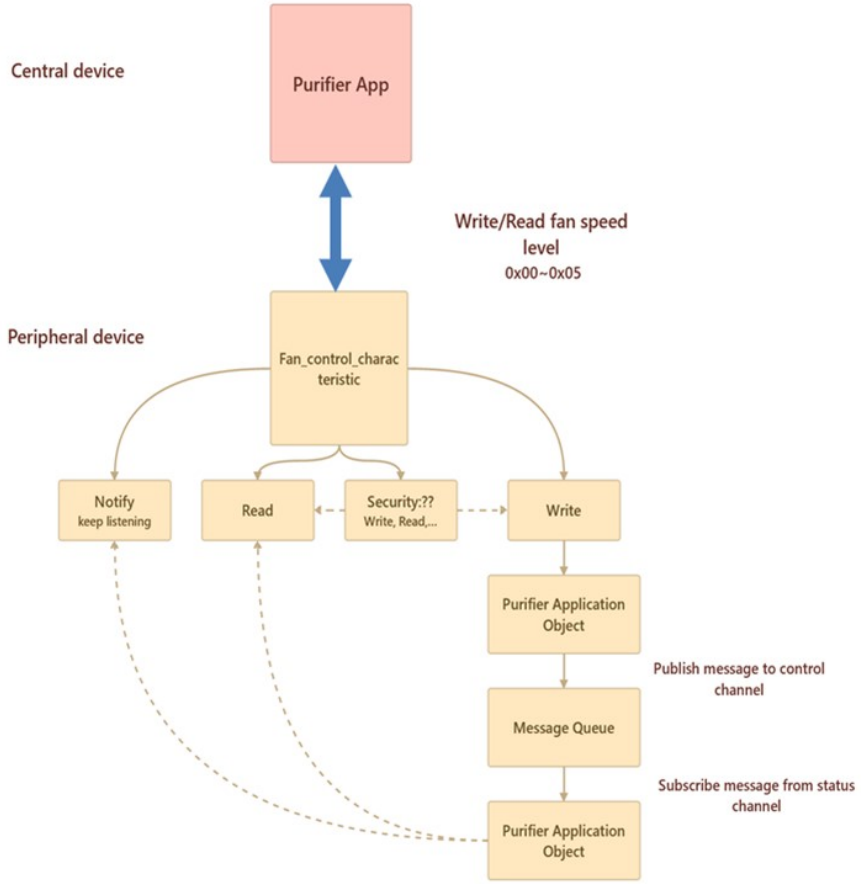

Figure 12. The working mechanism of one characteristic

4.2 **Optimization of status feedback**

There were 13 characteristics created for this BLE application. 10 out of 13 characteristics executed the normal read or write action to” Purifier” Object. As App on central device had to read all status periodically in seconds, all characteristics were called to fetch the latest status one by one. It generated amount of the repeated work and time during the process running.

4.2.1 Dictionary to status

The author used dictionary to manage the feedback from that message queue in a standard format. Dictionary is a special data type in Python. Unlike list, which is only accessed the position of its elements, dictionary could be accessed to its element via keys. When “Purifier” application part received the read or write request from any characteristic, there was a variable stored the string name related to the characteristic. When the status message was pulled out, that vector would be used to access the value in dictionary via the same key.

4.2.2 Quick response to all status notification

As central device did recycled status reading to all characteristics, there were many repeated actions to fetch the messages during the process. To simplify the process, as in Figure 13, the author set a period to expire for each time Central reading the messages from 10 characteristics. Before the time got expired, these characteristics only got the message fetched in the first second. In another way, the status messages had been decoded for the first characteristic calling “Purifier” application and utilized for the rest of other characteristics.

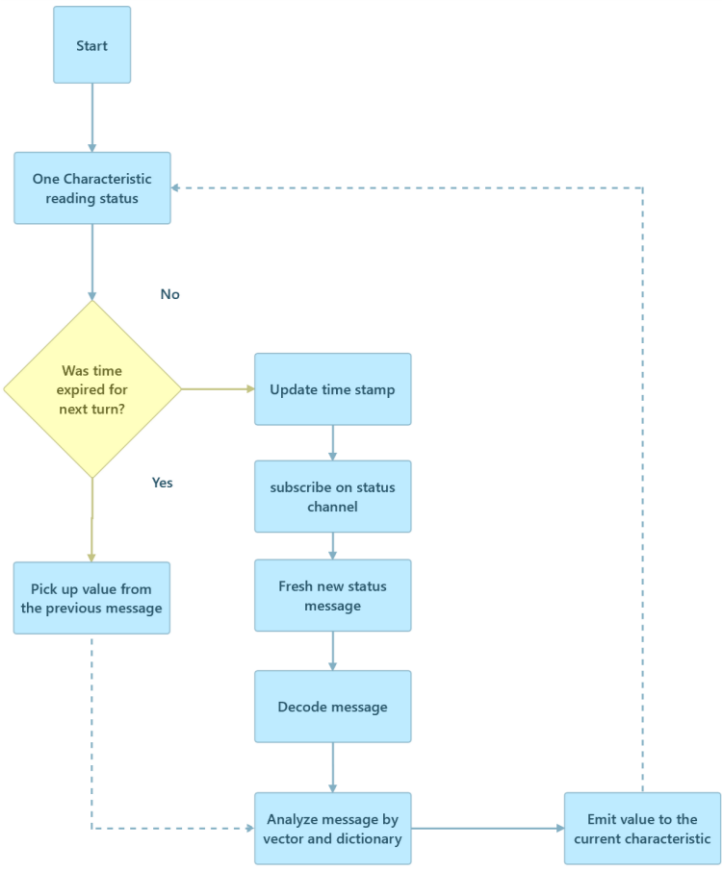

Figure 13. Cycle for message transfer to one characteristic

4.2.3 Long string protocol

To transfer the long string messages via Bluetooth, the programme was designed to split the messages string into packets which were sent from the Central device. Each packet was which were 20-byte long and got combined again into complete message on the side of peripheral device. At the first, Stream-Switch Characteristic opened the gate in the “Purifier” application for the first packet coming in and “Purifier” application emitted value “0X01” back to Stream switch characteristic. When the Central device read the hex value “1” from the Stream-Switch-Characteristic, it started to send the first data packet to String-Writing-Characteristic.

Since then, each time the central device started to transfer the rest of other packets; the “Purifier” application extended the string with the packets of strings coming. When string transfer finished, central device called String-Writing-Characteristic to close the gate, so Central device got hex value “0” from the Stream-Switch-Characteristic. Then Purifier” application loaded message to an ordered dictionary and computed the 32-bit CRC value of message.

After the calculation finished, the “Purifier” application compared the 32-bit CRC value in message and the computed result. It also compared the string length with the “length” element in the message. Then “Purifier” application classified the message string by its string type. Message was loaded and stored into the corresponding configuration files. Consequently, the “Purifier” application cleaned the result and emitted the hex value “0x00” signal back to the stream switch characteristic.

The author defined the string type to add different message into configuration files. The string type definition is as Table 6:

Table 6. String type definition

Index	String type	elements
1	cloud credential	uuid,network id ,token
2	timer program	schedule,modified time,created time uuid
3	wifi configuration	SSID,PSK

The example format of message string is:
{"Length":"171","CRC32":"908095D5","String_type":"01","Message":{"username":"xxxx@gmail.com","password":"xxxx","node_id":"xxxx","network_id":"xxxx","uuid":"xxxx"}}

4.2.4 WIFI configuration

To register the WIFI string and to force the device to use the new WIFI, the project used WPA_CLI as tool to manage the WIFI information. In the beginning, shown in Figure 14, the “Purifier “application got message via the long string protocol, meanwhile the string type is marked as 3. According to the string type, the related key “WIFI SSID” and “PSK” were picked out from the message dictionary, sent to the WIFI registration process.

During the WIFI registration process, the WPA_CLI class object received the WIFI SSID and PSK variable. It added a new network and returned a number.
Then WPA_CLI set SSID and PSK for the new network and sent out a configuration message in the configuration channel of redis client. After that, WPA_CLI select it as the solely active network right now. Finally, the “Purifier “application restarted the client process for connecting it to the cloud.

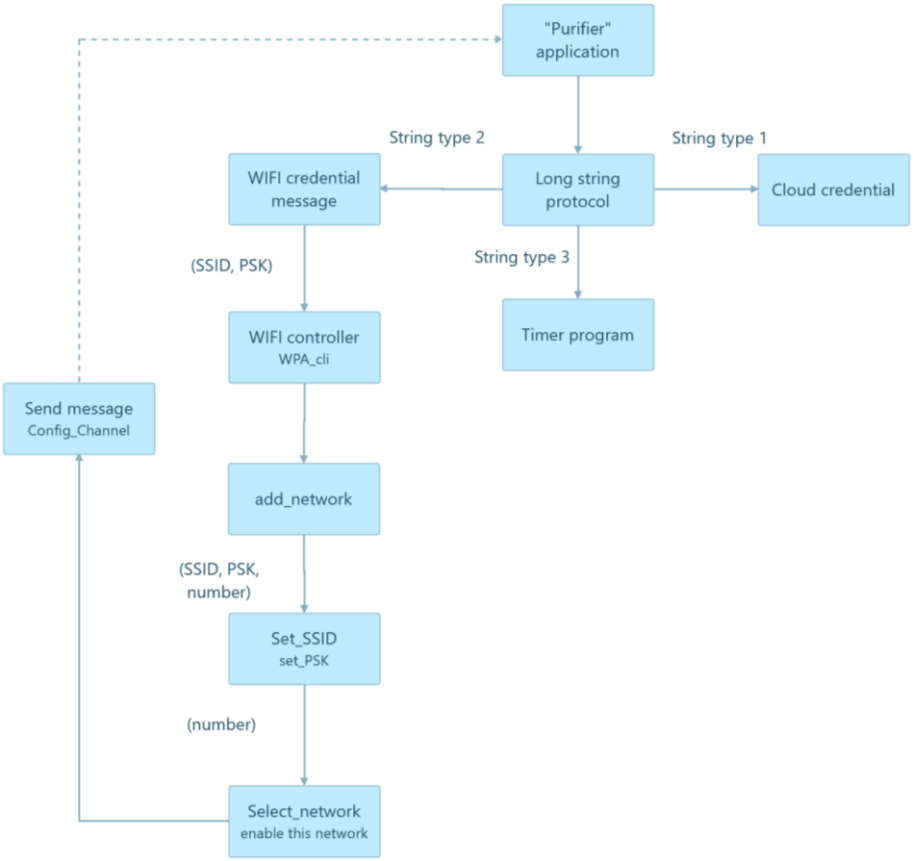

Figure 14. WIFI registration process

4.3 **Security measurements**

Due to the security demand, the BLE application needed hierarchy protection. To achieve it, the project plan to add a fixed password check in the mechanism of “Purifier”

object, an RSA encryption during the message transfer and the encryption for its pairing process.

4.3.1 Fixed password input

A password input was always asked by the process before any functional characteristic accepting the read or write request. The central should write password into “Password” characteristic which called the password checking function. It will compare the input to the fixed password string in the factory configuration file. After that, the long string protocol and other functions will be enabled to use.

4.3.2 RSA encryption

This project applied an RSA encryption method to protect the credential string transfer. The author and his workmates used “ssh_genkey” tool in the Linux system to generate a pair of key files. The public key was shared to encrypt the strings and the private key was kept as a secret file to decrypt the messages on the side of local devices. Considering the length of files transferred, the RSA length was selected as 3072-byte long. Figure 15 is an example of RSA 3072-byte key file.

```
"-----BEGIN RSA PRIVATE KEY-----\n\
MIIEpAIBAAKCAQEAmorSNhPdhS9c1bDZWxnNWgwGeC/cNYi/aw1v5HUFsGPvIgzk\n\
MuzTWBUDU8H36y9HL0M40RkuE3ib6CSQon1h2fOjNSiWnoal76ErfXmqejI2d+wR\n\
D7WAWJAjPVEvnNHlE+zkwMxuhCWv/Qx0tmyz2+j7R2CA4noYrBOofoIklCWhkrGT\n\
yerGIY+IcprGltUxTW6QFybKjH2HS2YSD34Fh+RhMy2d3St74osYQ0gg8omYpXd8\n\
xUhSsl1X0gCrJ0ZcBrsI1QzxQU2XFYyQvZBzNP8YkuQbxQ9woeYTPb7c6f52axWc\n\
zRl7GlZxCWwl/nXmY4lFxJJ4Lyp+h5m6xmO2RQIDAQABAoIBAFJnTTlalFbM72Je\n\
PzetZWrrWJPDaVb+SJsLeitGHrd0QfFVEGT7FKI16iVCGzDDb+CKpoQQOhpVyEQb\n\
IygRVRjDKbZRoyZ6Q4an2Km6MyWfGp4NOPe4XuxaRwAOByZGYTCjs0H+f7tk4Rc8\n\
CguvoKPU8g+YLIEWe4tdnA2A1YJMTkSfTO3j+8l1Vp9k9qOUxJbTnuy2pmVlJG/W\n\
3GDB6Q0WRlPuq8xGMeCS7Tk9eJGpQ50ySPN9Nt1hInO9k2SMjhonAfyyromPt+wi\n\
iv+kQwd0UqIsccw7u9QmnFeGxS3Z9oKcfHeHnCNOcXawQZxqYT1U5n93Wq2exc0F\n\
BaU2RaECgYEAzYTLe6fI5VGPg6SbBRB617/hZB4frwrJvHrnfme1amF/9LMukc1V\n\
jYhkAB1TL2qJkh2yHi/tsgYnu8okW8SplJ6m0G0/P0CdR6ZE3wd4rIMcMXN+2Gjf\n\
Cz9jomjYtMV+UnfI/EO0fCbEWBwXCBG6cfb6qCenbzSJE/Q5HXkKGu0CgYEAwICU\n\
exicBCf/I5qzE5s8XTRDWf2wqYgUQnPBRs2iR2Tmowgd70WZiTxh0cSixXryXkK/\n\
gBugmd64hdHeh2H311XBoNY+COKGypyZ8n/SSxp5Y4LT+mXEbY+tW+acrIPjgyFZ\n\
bLihdUkhuoiAlCLetYQQhnm0Po9ih1D6SGi6JbkCgYEAsUYEhysZHMUXd8P6e4nV\n\
jFhUwJlDVyeFD/Er0Vr/nz1us9IDL3zgeO7W40poQJQzCjhXNba93ztz9oXMiuO+\n\
KxYWad5yGctQX0dKTN8S9is5nGY1WdJL4lXRSXwKwRNBfT0Zvo9gFV8TKbVBIJK6\n\
MieJ2pYp1MCKJQ4aXhPT5ZkCgYEAhVVKAszz20kcxjtzh8QBFomJkP5Pa81OQwzM\n\
NU9e9FZmDUwk98LcaprBIjsxhtj2oOVYoNIB9jG22zGXEWyA7sx+NfnOs/a66AF5\n\
++Ye8Vgf9hX63ze3yy82Iy5jHHapkfz6ZHypkIQb8Yh0jqZsXYtgqtQHTUGhPBqR\n\
kK/BJzkCgYAfa38tW0hF50qX4jN4PuBWFSRhVum98OCxLXP0pplMldu8jnaDRRtT\n\
L4BGah/HKIPhDq8Kt+PFnWaZu8+eOmqz1dOkqLIj8dqd/4ZAqhP26r1PNPsjcCfX\n\
XL/V+TRXM0zc94pTKNJ7/95OfR58J1WV0z5x6K1QHkJK4Enq5X8Cwg==\n\
-----END RSA PRIVATE KEY-----"
```

Figure 15. RSA 2048-byte private key file example

The project used RSA python library to execute the decryption of messages. The “load_pkcs1” function in RSA library loaded the key text and converted it into a private key object. The message needed to be decoded as it was Base64 format and then

decrypted by the private key object. In this way, the message could be efficiently decrypted, despite the whole process still spent over 10 seconds.

4.4 **Pairing process implementation**

As the project was involved in the home automation, the project design should guarantee the security of Bluetooth link. The author intended to implement the security feature that BLE pairing and bonding in Pybleno. As the security modes and procedures were defined by GAP and based on Security Manager Layer, the focus of security implementation is on the Security Manager Protocol (later referred to as SMP), which is SMP python script file in “hci_socket” package of “Pybleno”. The algorithm SMP used to generate the confirm number, and STK in pairing process was cryptographic toolbox, which was a crypto script. The long-term key is generated by the current SMP file which only allowed BLE legacy pairing “Just works”. It triggered a window on the central device to customer as “Pair” or “Cancel” option shown as in Figure 21.

4.4.1 Pairing procedure

The security management started from the initialization, where it set the listening to the “Aclstream” object and executed the pairing process, encryption change, long-term-key negative reply. It caught the remote address of which the type was “Random”, and local address of which the type was “Fixed”. Remote address and local address both participated in the calculation of confirm number and random value. But remote address was also needed in the long-term key generation during the further bonding process.
All pairing operations between the central and peripheral devices were named with the hex code in Table 7:

Table 7. SMP pairing operation code

SMP_PAIRING_REQUEST	0x01
SMP_PAIRING_RESPONSE	0x02
SMP_PAIRING_CONFIRM	0x03
SMP_PAIRING_RANDOM	0x04
SMP_PAIRING_FAILED	0x05
SMP_ENCRYPT_INFO	0x06
SMP_MASTER_IDENT	0x07
SMP_UNSPECIFIED	0x08

In the Legacy paring phase 1, the peripheral device exchanged pairing feature with the central device. At the beginning, it was the central device to initialize the pairing, as an initiator role. The peripheral device acted as responder. The central device will send its pairing features as an array, for example: array ('B', [1, 4, 0, 45, 16, 11, 11]), with 1 at the head because it is pairing request code. After responder receiving the pairing request message from initiator starting with 0X01, it got the IO features as an array.

This array declared the IO capability as 0x03 because the responder device has no input function for users to type pin code or output function to indicating the pin code or status. The second one is OOB authentication flag, deciding the enabling of OOB mechanism used during pairing process. The OOB mechanism will be used if devices have extra authentication demand. In this project, the OOB mechanism was not used, setting flag as 0x00. For authentication requirement, this project intended to set bonding, while Just Work did not have MITM protection. The command set as 0x01 according to its format. The maximum key size was 10 octets. According to the Bluetooth 4.2 specification, the key length should be between 7 octets and 16 octets.

After the central device received the peripheral pairing features, it sent the confirm number calculated, for example:
array ('B', [3, 62, 68, 1, 207, 131, 60, 189, 239, 40, 186, 195, 147, 125, 176, 222, 37]). It is the pairing confirm number, with head code 3. When the peripheral received the message, it created its own confirm number by executing functions from the crypto script. The confirm number will be sent to the initiator device, which then sent the random number to the responder device. The slave device used c1 function again to calculate the initiator devices confirm number and compared it with the confirm received. If they are the same, responder device would create STK by function s1 from crypto script. Then it sent the random to initiator device.

Meanwhile, the long-term key(ltk) was created and written by utilizing function “addLongTermKey” from “mgmt” script. This function required input: the remote address received, remote address type, authenticated flag, initiator, ediv, rand, stk.

4.4.2 Cryptographic toolbox

For the current BLE legacy pairing method, the cryptographic toolbox only contains the functions c1 in Formula 5 and s1 in Formula 6 to generate confirm number and STK during the pairing process.

$$\begin{aligned} confirm\ number = \ & c1(temporary\ key, \\ & Random\ Number, \\ & Pairing\ Request, \\ & Pairing\ Response, \\ & master\ address\ type, \\ & master\ address) \end{aligned} \tag{5}$$

$$ST = s1(TK, slave\ random\ number, master\ rand\ number) \quad (6)$$

As 1 byte is 8 bits, the temporary key (later referred to as tk) was 16 bits long, Random Number was 16-byte long random number from crypto library. The Pairing Request command and $Pairing\ Response\ command$ were string with 7 bytes. The c1 function required security function e to encrypt the 128-bit data with an AES-128-bit block cypher, and the logical XOR gate function to execute XOR logical among elements. The equation is:

$$c1 = e(k, e(k, r\ XOR\ p1)\ XOR\ p2) \quad (7)$$

where

$$p1 = pres \,||\, preq \,||\, rat' \,||\, iat' \quad (8)$$

$$p2 = padding \,||\, ia \,||\, ra \quad (9)$$

where rat' is 8-bit with rat at the least significant bit, and iat' is 8-bit with iat at the least significant bit.

s1 function required 3 inputs:

- 128-bit k: the temporary key defined by SMP.
- 128-bit r1: the random number responder used to generate confirm number.
- 128-bit r2: the random number initiator device used to generate confirm number.

The equation of s1:

$$s1(k, r1, r2) = e(k, r') \quad (10)$$

where:

$$r' = r1 || \mathrm{r2} \quad (11)$$

5 DEBUGGING PROCESS

The Bluetooth application were debugged by testing the pairing process and the read or write of functional characteristic. A test app the author used was Light Blue installed on iPhone. The commissioning company has its own app designed for this BLE application. The public Light Blue was used to test the function and encryption of the BLE project in this thesis.

5.1 **Pairing debugging process**

The pairing process could be debugged by adding a “secure” feature on characteristics like fan speed control and then restarting the whole process. At first, test app Light Blue scanned the Bluetooth devices nearby and found the device name “Purifier” shown as in Figure 16. User could press this peripheral name and enter the service page to find more characteristics.

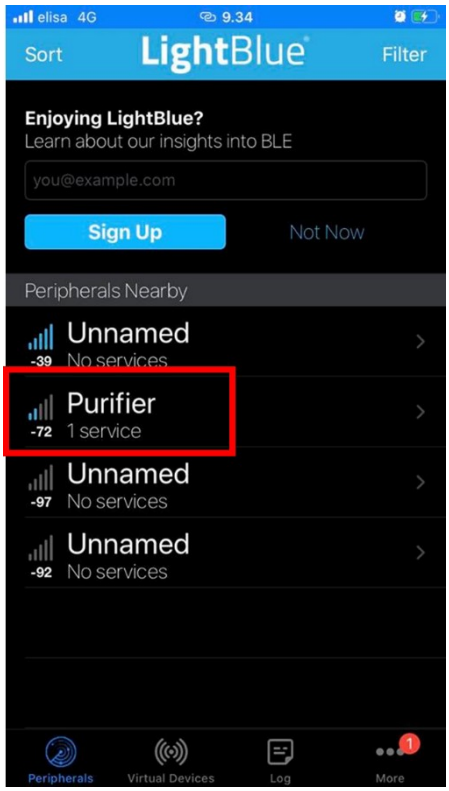

Figure 16. Scant list on App Light Blue

After entering the service page, it presented the UUID of Serivice, and the list of characteritics implemented as in Figure 17. Each characteristic had its own 16 -octet UUID number, and properties as Read, Write or Notifiy. All characteristics UUID were called from external configuration file from “0X455410d6fa5548a0aae0bd7a8fe12101” to “0X455410d6fa5548a0aae0bd7a8fe12113”. The characteristic with UUID end “04” was the Fan-Speed-Control, and marked with “secure” feature.

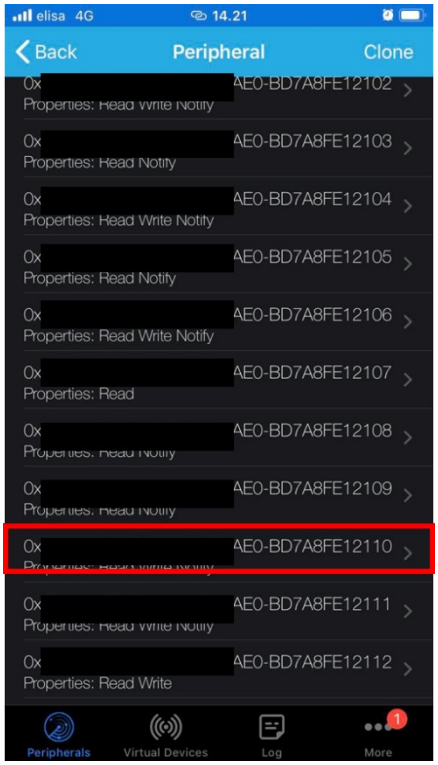

Figure 17. Service page of "Purifier" peripheral

Meanwhile, the SMP in HCI socket package was initialized, to capture the remote device address "46:40:5F:96:18:0E". Its address type was random. The local BD address was "B8:27: EB:9B: 57:9A" and its address type was fixed. The remote and local address were reversed to move their most significant octet to the left.

In Paring phase 1, the initiator device sent pairing request with its pairing features to responder device as $[1, 4, 0, 45, 16, 11, 11]$. It means that the initiator device had keyboard display, with no OOB flag to demand authentication. The initiator device required MITM protection, supporting secure connection, and the maximum encryption key size is 16-octet, with the responder key distribution.

Slave device responded to the request by sending its own pairing feature as $[2, 3, 0, 1, 16, 0, 1]$. It means the responder device had no input or output function, without OOB authentication. The slave device required bonding, with maximum encryption key size 16-octet.

After exchanging the IO feature, the pairing process went to the phase 2 to calculate the confirm and generate the short-term key for further bonding process. Slave device would receive the confirm number first from the initiator device, for example [3, 75, 65, 170, 122, 52, 228, 26, 211, 250, 181, 70, 213, 161, 154, 240, 44].

After that, the responder device began to calculate its own confirm number. According to the definition of c1:

$$p1 = pres \,||\, preq \,||\, rat' \,||\, iat' \qquad (12)$$

Where:
$preq = [1, 4, 0, 45, 16, 11, 11]$
$pres = [2, 3, 0, 1, 16, 0, 1]$
$rat' = 0X00$
$iat' = 0X01$
$p1 = [1, 0, 1, 4, 0, 45, 16, 11, 11, 2, 3, 0, 1, 16, 0, 1]$

$$p2 = padding \,||\, ia \,||\, ra \quad (13)$$

Where
$padding = [0,0,0,0]$
$ia = [14, 24, 150, 95, 64, 70]$
$ra = [154, 87, 155, 235, 39, 184]$
$p2 = [154, 87, 155, 235, 39, 184, 14, 24, 150, 95, 64, 70, 0, 0, 0, 0]$

When
$k = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$
$r = [3, 239, 131, 35, 55, 106, 18, 153, 9, 46, 137, 98, 101, 237, 54, 99, 3]$

According to the Eq.7:
$c1 = [239, 131, 35, 55, 106, 18, 153, 9, 46, 137,98, 101, 237, 54, 99, 3]$

The confirm numbers sent by purifier as responder device was [3, 239, 131, 35, 55, 106, 18, 153, 9, 46, 137,98, 101, 237, 54, 99, 3] with the first octet 3 as the operation code. When the initiator device received the confirm numbers from responder device, it sent out the random value used in previous c1 function: [4, 7, 109, 38, 96, 155, 37, 93, 89, 19, 121, 223, 126, 232, 12, 108, 238]. Slave device used this random number to calculate the initiator confirm number and compared the calculation result with the confirm number received before. The calculation used c1 again with the same temporary key, the random number from initiator device, the initiator address info and responder address info. Once the calculation result was the same as the number received, the responder sent out the 16-octet random number utilized in c1 function to the initiator for confirm number calculation. If the result did not match the initiator confirm number, the responder device would send operation code SMP_PAIRING_FAILED, SMP_PAIRING_CONFIRM, and terminate the pairing process.

Then the pairing process generated the short-term key by the function s1.
As

$$Short\_term\ key = s1(temporary\ key, responder\ random\ number, initiator\ random\ number) \quad (14)$$

According to the “Specification of the Bluetooth System”, the definition of s1 is:

$$\mathrm{s1(k, r1, r2)} = \mathrm{e(k, r')} \quad (15)$$

where

$$r' = r1' \,||\, r2' \qquad (16)$$

The r1' is the most significant 64 octet of r1, and r2' is the most significant 64 octet of r2. For example:
if
$\mathrm{r1} = [4, 7, 109, 38, 96, 155, 37, 93, 89, 19, 121, 223, 126, 232, 12, 108, 238]$
$r2 = [3, 239, 131, 35, 55, 106, 18, 153, 9, 46, 137, 98, 101, 237, 54, 99, 3]$

$r1' = [4, 7, 109, 38, 96, 155, 37, 93]$
$r2' = [3, 239, 131, 35, 55, 106, 18, 153]$

then
$s1 = [99, 152, 192, 18, 248, 195, 8, 15, 195, 149, 174, 133, 102, 148, 215, 247]$

Then the pairing process entered the phase 3 to add the long-term key. The current Pybleno library doesn't have the protocol file to support this mechanism, so the implementation of long-term key could not be achieved.
With the above debug test process, this BLE project achieved the simplest pairing between the initiator device (commissioner app) and the responder device(purifier). But the development failed to build the bonding process due to the limitation of knowledge and technical conditions. The author needs to do further study in the BLE pairing and encryption development.

5.2 **Function debugging process**

The function debugging test required to test the status reading and writing of characteristics such as fan speed, automatic mode, charging mode, cover open status, reset press, power on switch, series number, password input and so on.

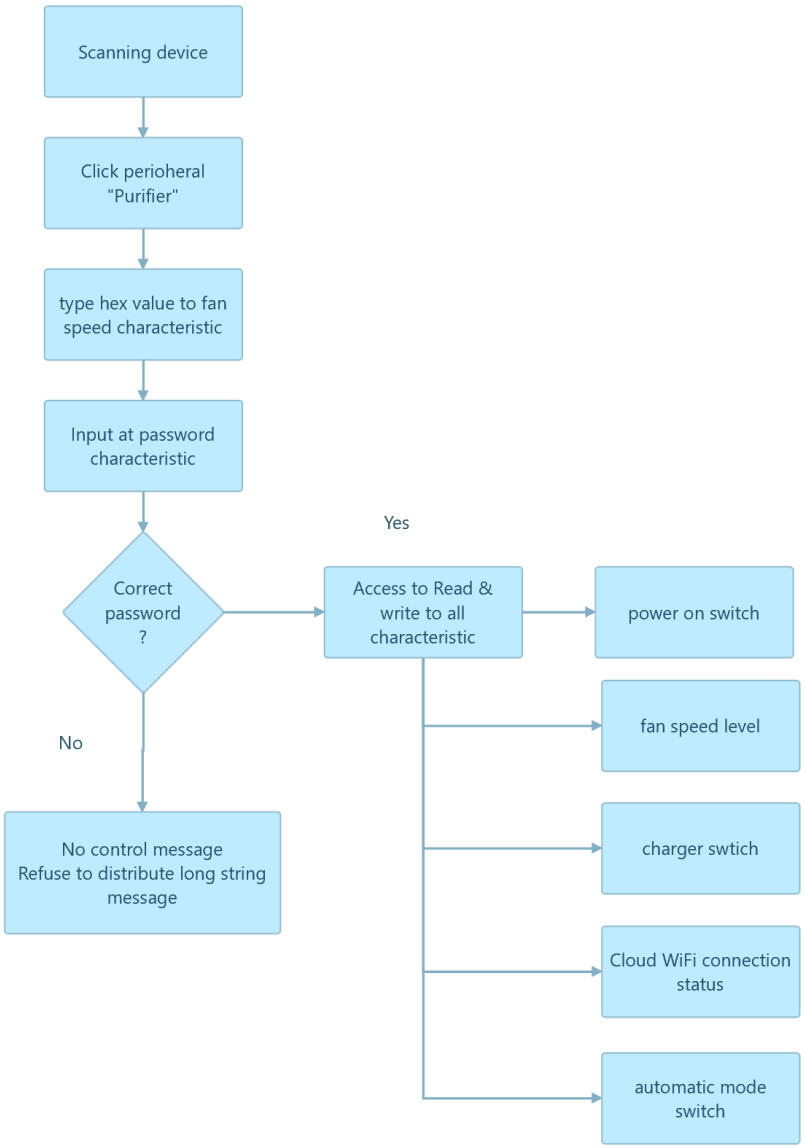

Figure 18. BLE application debugging process design

Shown as in Figure 18, firstly, the Light Blue found the device advertising nearby and displayed the name “purifier”. As the original designed, a customized app should input the password into the password characteristic.

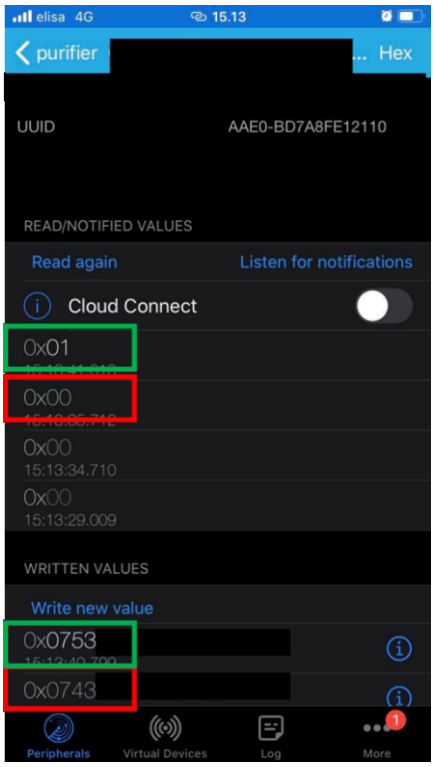

Figure 19. Password characteristic

As Figure 19 shows, once the characteristic received the wrong password, it returned the value 0x00 to status list. When the input numbers to password characteristic was correct, the characteristic return value 0x01 to status list. After that, all characteristics of the service in the device “Purifier” accepted the reading and writing request. In the Chapter 4 “Application foundation”, the second section “Optimization of status feedback” implement the quick response to all status reading when the customized app intended to read all characteristics status. Within pre-set time period, all characteristics would only feedback the value which was fetched from the first second.

Next, the customized app should read status from the fan speed characteristic and write new values to change the fan speed. User would be asked to pairing device or not. Choosing “Cancel” would disconnect the purifier immediately. Instead, user should choose “Pair” and set the characteristic keep listening to notification, shown as in Figure 21. After that, the customized app could type different fan speed values to “Write” function and the characteristic would return the current value after it received the control message, shown as in Figure 21.

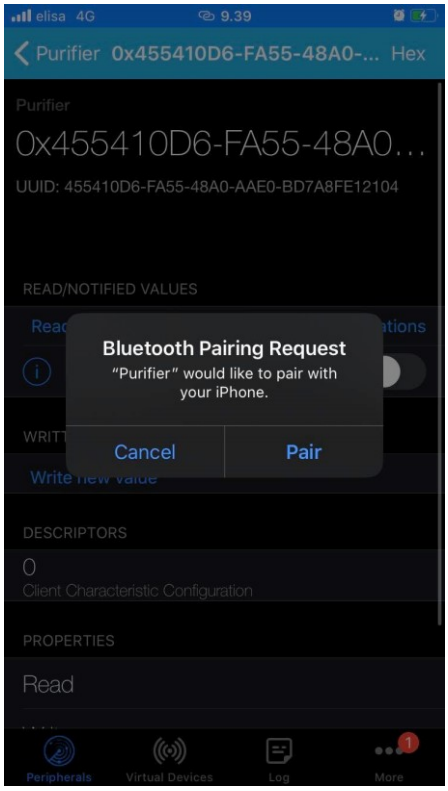

Figure 20. Pairing request

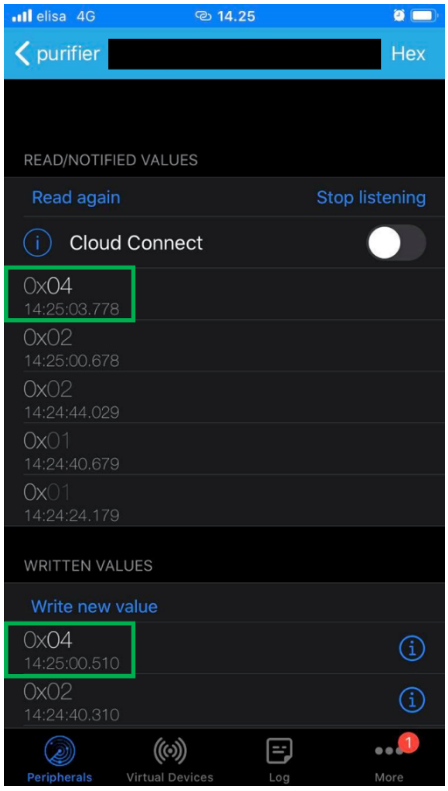

Figure 21. Fan speed characteristic page

Other characteristics such as an charging mode switch, an automatic mode switch, a power switch accepted hex number such as 0x01 and 0x00 because these characteristics would send the boolean value to the control message in the “Purifier” object. Through the debug test of password input and fan speed control, it already showed that the this BLE project was able to interact with the Light Blue App. This BLE project can receive control message from App and respond latest status to App.

6 CONCLUSION

The purpose of this project was to explore a Bluetooth Low Energy application with microcontrollers for the beginning developers. It could be used as an implementation to the design projects of automation students at Häme University of Applied Sciences.

The BLE application was part of the local wireless control in the purifier project. The author was commissioned to provide a local control solution for a stand-alone device. It was required to be quickly set up and stable in communication. During the development the author got familiar with the Bluetooth Low Energy protocol and had a chance to test the “End to End” principle in a real project. The concept of an End to End design is to set margins of application to the end node of one network. Its mechanism is like package design which specifically focuses on a single function. End to End also provides an overview to project design and enforces the developers of each distributed region to cooperate.

The author needed to design the BLE application to meet the requirements of the commissioner, and its compatibility with other blocks such as cloud clients, and local control hubs. The development work required a high amount of time on debugging issues on input and output performance, for example message decryption errors. There is also space in the process mechanism to be optimized because redundancy in the mechanism would slow down the process reaction. To accelerate the function response, the author tried to remove the repeated action to fetch the message and he used dictionaries instead of if-else condition sentences to pick up the values needed.

Still, there is a lot of work needed to enhance the security of the pairing method in this Bluetooth project. The debug test only relied on the test application and command prompt outputs from the Raspberry Pi 3b board. The Bluetooth sniffer tool will be equipped as a technical assistance tool in the future. With sniffer tools, it will be possible to catch the advertising and data packages transferred in the air. The pairing and bonding feature construction requires more knowledge on the link layer and the physical layer at the Bluetooth controller level. The author needs to learn more about function and algorithm to find an alternative solution here because the author is interested in complementing the current Pybleno library in future projects.

The author is deeply grateful to the commissioning party for the company’s instrumental support and to HAMK’s supervisor for giving suggestions on both academic writing and his participation during the debugging test.

REFERENCES

Afaneh, M. (n.d.). *The Basics of Bluetooth Low Energy (BLE)* . Retrieved from NovelBits: https://www.novelbits.io/basics-bluetooth-low-energy/

Bluetooth low energy Security Fundamentals. (n.d.). Retrieved from TEXAS INSTRUMENTS: http://dev.ti.com/tirex/content/simplelink_academy_cc2640r2sdk_2_30_02_00/modules/blestack/ble_02_security/ble_02_sec_basics.html

Developer Study Guide: An introduction to Bluetooth Low Energy Development . (2018, December 17). Retrieved from Bluetooth: https://www.bluetooth.com/wp-content/uploads/2019/10/BluetoothLEDeveloperStudyGuide_V5_1_1.zip

Introduction to Redis. (n.d.). Retrieved from redis: https://redis.io/topics/introduction

Langley, A. (2017, 11 28). *Adam-Langley/pybleno*. Retrieved from Github: https://github.com/Adam-Langley/pybleno

Lindh, J. (2016, October). *Bluetooth® low energy Beacons.* Retrieved from TEXAS INSTRUMENTS: http://www.ti.com/lit/an/swra475a/swra475a.pdf

Microchip Technology, I. (2019). *Bluetooth® Low Energy Packet Types*. Retrieved from MICROCHIP Developer Help: https://microchipdeveloper.com/wireless:ble-link-layer-packet-types

Raspberry Pi 3 Model B. (n.d.). Retrieved from Raspberry: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/