Implementation of a Portable Network Using LPWAN's LoRa Technology

By

Heleen Aras Ahmad-Saib

University of Kurdistan Hewlêr

Erbil, Kurdistan

BSc

July 2024

Implementation of a Portable Network Using LPWAN's LoRa Technology

By

Heleen Aras Ahmad-Saib

Student Number: 01-20-00177

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Engineering

Department of Computer Science and Engineering

School of Science and Engineering

University of Kurdistan Hewlêr

BSc

July 2024

Erbil, Kurdistan

Declaration

I hereby declare that this dissertation/thesis entitled: “Implementation of a portable network using LPWAN’s LoRa technology“ is my own original work and hereby certify that unless stated, all work contained within this is my own independent research and has not been submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Signature

Name: Heleen Aras Ahmad-Saib

Date:

Supervisor’s Certificate

This dissertation/thesis has been written under my supervision and has been submitted for the award of the degree of Bachelor’s in Computer Engineering with my approval as supervisor.

_____ _____

Signature Name

_____

Date

I confirm that all the requirements have been fulfilled.

_____ _____

Signature Name

_____ _____

Head of Department of Date

I confirm that all the requirements have been fulfilled.

_____ _____

Signature Name

_____ _____

Dean of School of Date

Examining Committee Certification

We certify that we have read this dissertation/thesis:

……

and as a committee have examined the student:

……..

in its content and what is related to it. We approve that it meets the standards of a dissertation/thesis for the degree of BSc in Computer Engineering

_____	_____
Signature	Signature
Name:	Name:
Chairman	Supervisor
Date	Date

_____

Signature

Name:

Member

Date

Dedication

I dedicate this work to all the ambitious dreamers, who are lingering between their imagination and broken faith, you can do it, just keep moving forward.

Pray hard, and work harder.

Helin Aras Ahmad-saib

14th July 2024

Acknowledgments

To begin with, I would like to express my gratitude to my supervisor, Dr. Govand Kadir, the one who motivated me, and constantly believed in me. I also want to thank my family members, the supporting columns of my life. Mama, my inspiration, you are the greatest female figure I have ever seen, and to be the woman you are is a dream I will always chase after. Baba, my best friend, and secret keeper, your embrace has always been my safe space when I'm lonely and scared. Shang, the other half of my soul, and my companion in life. Ahmad, Jwan, and Rand, you are the colours of my life, my love for you is endless.

This acknowledgment is also dedicated to my lifelong friends, Mannar, Haleen, and Esraa, the ones who stayed by my side in my darkest days and witnessed my bloom with eyes full of pride and love.

As a dormitory graduate as well as a university graduate, I would like to show my gratitude and love to my dorm mates. These are my partners through the thick and thin. I could not have possibly survived the four years of dorm without them, and I would like to express my utmost love and gratitude to Marwa, you were my life partner in these years, thank you for all the beautiful memories you added to my life.

Lastly, I am forever grateful to the beautiful city, Hawler. This city turned the 18-year-old girl who was naïve, and passionate into a confident, mature, and independent girl. Thank you for all the opportunities you gave me and all the experiences you taught me.

Abstract

The availability of network coverage is still a goal that is not achieved in many locations. People living, working, or even touring in remote areas with no network infrastructure struggle to communicate to this day. As the deployment of network towers and gateways might be overly expensive, other solutions like private networks or infrastructure-less networks that are based on communication between nodes with no gateways have been gaining popularity. Low-power wide-area networks (LPWANs) have been emerging as one of the leading technologies in making such systems. The properties of the LPWAN such as low-power consumption, long-range connectivity, and other qualities make it the ideal choice for the proposed systems. This project solves this issue by developing a messaging system based on the LPWAN technology LoRa. The project builds a mobile application that allows users to send text messages that are sent to a LoRa module connected to the device and transmitted to the intended receiver. The LoRa module will be based on security and the necessary data communication protocols, and therefore, it will become a reliable source of communication when other networks are unavailable.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

IoT	Internet of things
M2M	Machine to machine
LPWAN	Low-power wide-area network
LoRa	Long-range
ISM	Industrial, scientific, and medical
CSS	Chirp spread spectrum
WLAN	Wireless local area network
DBPSK	Differential binary phase-shift keying
GFSK	Gaussian frequency-shift keying
NB-IoT	Narrowband-IoT
GSM	Global system for mobile communication
LTE	Long-term evolution
SCFDMA	Single-carrier frequency-division multi access
OFDMA	Orthogonal frequency-division multi access
RF	Radio frequency
Bps	Bits per second
SF	Spreading factor
FFT	Fast Fourier transform
PHDR	Physical header

CRC	Cyclic redundancy check
CR	Coding rate
BW	Bandwidth
R_S	Symbol rate
R_c	Chip rate
FEC	Forward error-correction
GPIO	General-purpose input/output
ADC	Analog to digital conversion
DAC	Digital to analog conversion
SPI	Serial peripheral interface
I^2C	Inter-integrated communication
EU	European Union
GPS	Global positioning system
IDE	Integrated development environment
D2D	Device-to-Device
RSSI	Received signal strength indicator
TTN	The Things Network
STT	Successful transmit time
DTN	Delay-tolerant network
USB	Universal serial bus
MTU	Maximum transmission unit

Chapter 1 Introduction

The evolution of various network technologies has made it possible for digital devices to communicate with each other. Decades ago, before the emergence of data transmission networks, computers and other digital devices were isolated units that had no means of communication with each other. The batch processing system, in which batches of cards punched with holes were manually transferred to the receiving device was one of the earliest forms of data transmission (Green, 1982), in which this process was time-consuming and prone to error. Furthermore, in August 1962, for the first time, data was delivered through a network in the form of a series of memos (Leiner et al., 1997). Subsequently, network systems have been constantly evolving. Particularly, the development of wireless communication has been essential due to the need for data reachability (anywhere) and mobility (portable network). Therefore, many wireless data communication technologies have been developed so far, such as wireless Wi-Fi, cellular data networks, satellite communications, Bluetooth, and others. Because of all these technologies, we can see the rapid growth of mobile devices (telephone and network) and the Internet of Things (IoT).

The breakthrough of IoT technology has been revolutionary in the field of data communication and networking. A survey reported on the Forbes website forecasts more than 75 billion IoT device connections by 2025 (Chaudhari and Zennaro, 2020. P.1). IoT stands for any device like, sensors, actuators, software, and other technologies that exchange data with other devices over the Internet or other communications networks (Wikipedia, 2019), therefore becoming “smart devices”. These smart devices can interact with end-users and other smart devices. Interaction between smart devices is normally known as machine-to-machine (M2M) communication. Therefore, M2M is a type of IoT since it involves device communication across a network, and hence can be interchangeably used with the term IoT (Bembe et al., 2019). IoT devices are usually Battery-powered systems and therefore require low power consumption. Since the emergence of IoT, it has been used with many wireless networks such as wireless Wi-Fi, ZigBee, Bluetooth…. etc. However, these networks either have high power consumption or short-range coverage.

A promising class of wireless communication is the low-power wide-area network (LPWAN). As the name suggests, LPWAN offers a long-range, low-power network with low cost as well as low data rates. These qualities make LPWAN an ideal choice for battery-powered devices that require long-range with low power consumption. LPWAN can be seen as superior to other short-range wireless technologies for various use cases, especially in IoT applications (Raza et al., 2017). A rising technology of the LPWAN is the LoRa modulation. LoRa (long-range) is a physical layer technology that operates in the unlicensed industrial, scientific, and medical (ISM) band, and sends data using a spread spectrum modulation referred to as the chirp spread spectrum (CSS) (Raza et al., 2017). A promising solution is given by LoRa technology which provides both long-range and low-power specifications at the cost of low data rates. These qualities make the LoRa a perfect choice for IoT systems.

Traveling to remote areas requires a mobile network that can be carried anywhere, and therefore not restricted to any fixed coverage range. This will access communication between users of the same network no matter where they travel. Additionally, work affiliated with organizations, companies… etc. requires a fast network connection that is private to the entity only, and so, data can travel as fast and as securely as possible.

Therefore, this project proposes an IoT-based system solution that configures a mobile and private point-to-point (P2P) network using LoRa technology. Furthermore, the network configuration should include some of the primary networking principles like security, data packeting, addressing… etc. Configuring a private and mobile network is a solution to many cases where a fast and reliable network is essential and cannot be guaranteed using local networks. Thus, this project aims to create a solution by configuring a mobile private network using LoRa communication. According to Semtech, the manufacturing company of LoRa chips, LoRa offers long-range communication up to 15 kilometres in line of sight in rural areas. LoRa chips also have low-power requirements and therefore can work on battery systems efficiently. This makes LoRa transceivers an ideal choice for this project.

The LoRa board will be connected to a mobile phone via Bluetooth BLE where a messaging application will be developed as part of the project to enable interface with the LoRa system. The proposed project establishes a private and portable network that can be used anywhere and is not restrained by limited coverage. In addition, the system operates on the unlicensed free ISM band and therefore does not require subscription fees, along with the fact that the network is private, it will be much faster and more reliable because there will be little traffic and transmission distance.

1.1 Problem Statement

Communication in Rural and remote areas with weak or no signal at all can be very difficult. Similarly, in areas with increased data traffic, the network speed decreases, eventually creating an unstable connection. Hence, a private network is essential to have in these cases. Data communication techniques like two-way radio communication and satellite communication are vastly used for communication in remote areas. Examples like walkie-talkies and satellite messengers are among the most used commercial devices for this purpose. However, two-way radios i.e., walkie-talkies, can only communicate with voice, have short-range communication, and most of them do not provide security. Satellite messengers come with drawbacks as well for they take a lot of time to send or receive a message (up to 2 minutes) because of the distance between the satellite and Earth.

1.2 Objectives

Development of an IoT system based on LoRa technology by configuring a point-to-point private network with LoRa transceivers, and subsequently adding features like security by applying an encryption algorithm. Finally, creating a mobile application as an interface for users to communicate with through messaging. Below is an illustration to better describe the objective.

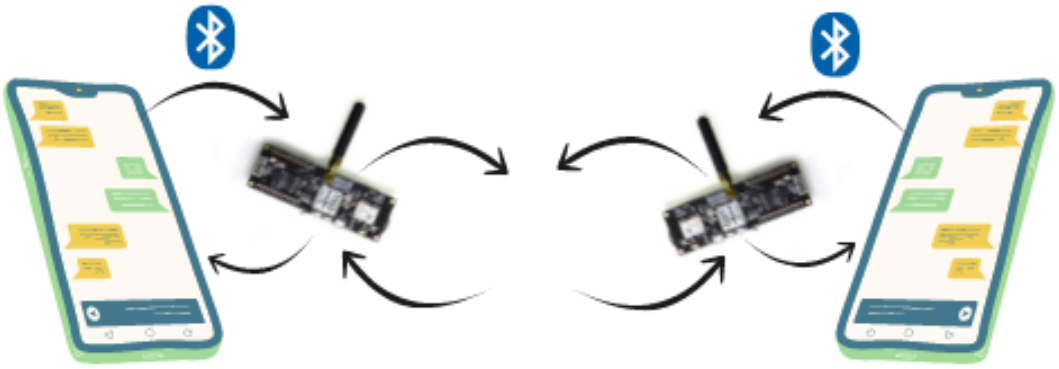


Figure 1.1 Overview of the project

1.3 Thesis Organization

After providing a brief understanding with the introduction, the following chapter will be specifically focused on providing an insightful understanding of the LPWAN network generally and LoRa technology especially. Chapter 3 is Literature review, where relevant articles and research on LoRa will be reviewed and briefly explained. Chapter 4 is implementation, that will thoroughly go into every step of the development of the project. Chapter 5 is testing, where the tests done through the implementation process are presented and explained. The last chapter is the conclusion of this thesis.

Chapter 2 Literature Review

The rise of IoT has expanded the demand for increasing network capacity. As the number of IoT devices rises in various fields such as agriculture, smart systems, industries, and data communication, the debate over finding efficient solutions for wireless data communications grows more and more. Consequently, systems built in remote areas lack proper network coverage because of the absence of cellular network infrastructure. Additionally, as the deployment of cellular towers can be quite expensive, new solutions have come to the surface. A promising technique is Device-to-Device (D2D) communication, which is a term used to describe the direct data communication between two mobile users without transferring through base stations or gateways.

Many research studies have been made that implement different technologies that establish network communications between devices.

One of the earliest wireless communication methods is satellite communication which is assumed to be able to cover the whole planet. Sarwar and Dempster (2009) introduced the SPOT satellite messenger, a device that gets its location coordinates from the GPS satellite network and sends messages to the Global Star commercial satellite constellation which then is sent to its earth stations and forwarded to the user's profile. This device is proposed to be used for tracking, messaging, and emergency-seeking. However, tests showed that SPOT has a reliability of approximately 40% within 1200 seconds, mostly because it was unable to penetrate through rooftops and therefore could not reach indoor test points. Other studies rely on mobile device D2D communication networks like Wi-Fi Direct and Bluetooth. El Alami et al. (2017) use Wi-Fi Direct to establish a connection, and then forward it to the cloud. The device receives a packet, if it doesn't have internet access, it re-broadcasts data to other devices, and if it has access, it will forward data to the cloud. The system is based on two protocols, multicast DNS and DNS-SD which is an extension to the first. The system is tested on three devices with a fixed distance of 50m, with vague results showing only the time response of each. Li et al. (2020) also established a connection with Wi-Fi Direct. They provide tests on both Inter-group and Intra-group Wi-Fi Direct. Results for

Intra-group communication show a throughput of 20.9Mbps, goes below 10Mbps after 30 meters, and almost fades away at 42 meters. Wi-Fi is a short-range network, and this explains why the distance results are so low. Therefore, LPWAN-based systems are growing exponentially, they solve the problem of short-range networks by providing long-range, low-power connectivity. In a study on *Sigfox* technology conducted by Mikhaylov et al. (2020) where they used two mechanisms, packet repetition, and multi-gateway reception. The test was carried out in 311 different locations. Results showed that delivered packets from 297 locations, mostly within 0.7 to 2 km from the nearest gateway got a packet delivery ratio of 94.79% of total locations. Tsavalos and Abu Hashem (2018) implement a testing of Sigfox's *Pycom* module. Their test includes various scenarios as indoor, outdoor, urban, and rural over 19 different locations. Results measure parameters such as path loss, latency, outage capacity, and Received Signal Strength Indicator (RSSI) values. Latency was between six and nine seconds and a maximum range of 14km.

Another LPWAN technology is the LoRa which is intensely used in various IoT systems due to its high performance in data rates, latency, scalability …etc. Centenaro et al. (2016) deployed a LoRa private network in a building to read temperature and humidity levels. The network consists of 32 LoRa nodes over 19 floors with one gateway on the ninth floor, and lastly, a server *NetServer* that bridges the monitor and database. As stated by the authors, the network had been working for a year till the time the paper was published and had passed all tests successfully by covering a radius of 2km however, this is only possible by using the minimum data rate modulation which also provides maximum robustness. The study estimates further expansion of LoRa nodes and claims that a single LoRa gateway can reach up to 15000 nodes in an urban area. Moving to rural implementations of LoRa, Prakosa et al. (2021) apply LoRa to smart agriculture. Data required such as temperature, humidity, PH levels …etc. are sent from sensors via LoRa nodes to a LoRa gateway which forwards it to the server where it can be reached by end-users. The first test showed an average of only 145-260ms for a distance of 400m and a spreading factor (SF) of 7, 250ms for an SF of 9, and 1320ms for an SF of 12. The maximum distance reached was 1km for an SF of 12. Further studies

were conducted by Bouras et al. (2019) who developed a wearable end device connected to a LoRa device that is worn by users with special needs to monitor their location and physical status. The LoRa chip in the wearable sends data to a nano-gateway which directs it to the Things Network (TTN) server. The study compares this method by using Wi-Fi instead of LoRa and concludes that LoRa is an ideal choice as it's battery-efficient, and provides a longer range, without providing any numerical data. LoRa is put under the test in another study which was experimented on by Kulkarni et al. (2019) where he distributed several LoRa nodes over a university campus in different places outside and inside, at the line of sight, and with obstacles. Each device transmits 100 LoRa packets with ACK. The study measured time, packet delivery ratio, and RSSI value of all nodes. In locations with no clear line of sight, the best-achieved distance was 500m. Additionally, a high packet delivery ratio was observed for almost all nodes with different spreading factors and coding rates.

Another rising use case of LoRa is for message exchange between end users. This is a crucial topic as was mentioned at the beginning of this chapter some technologies that are used for messaging between users, nevertheless, they come with drawbacks such as latency, power efficiency, and coverage. Thus, much research has been published on how to solve these issues with LoRa. Cardenas et al. (2020) developed a messaging application with LoRa network that serves as a solution for communicating in areas with weak or no network coverage. The authors argue that LoRa is a much cheaper alternative instead of deploying the infrastructure of cellular networks. The system is composed of *hubs* that provide both Wi-Fi and LoRa connectivity. End users are connected to hubs with Wi-Fi which should be in close range of the *hubs*, and the *hubs* are connected by LoRa over a wider distance. A web-based interface is developed to allow users to type messages. Every user needs to be registered to identify the node before being able to send messages. Messages can be sent to individual users or broadcast. Furthermore, a simple stop-and-wait ARQ protocol is used for packet transmission reliability. The testing of the system measured the user discovery process which was an average of 2.077msec, and successful transmit time (STT) which measures the total time for packet delivery. The STT showed almost constant behaviour with in-

creasing distance but varied for different packet sizes. Overall, the system was able to send packets at a distance of 6.9km in a clear line-of-sight route. Another study by Gambi et al. (2018) uses LoRa for smart home appliances. In this research, the sensors and actuators are connected to a LoRa module which is embedded in a gateway that uses a processing unit to convert the LoRa data packets to the publisher-subscriber message format of the MQTT server. Therefore, the data is ready to be sent to the MQTT server via Wi-Fi and can be accessed from the user's mobile phone that is subscribed to the MQTT server. LoRa nodes were placed both outside and inside, same floor or on different floors. Almost all scenarios had a success rate above 90%. Höchst et al. (2020) use LoRa modules to create D2D communication between mobile devices in the absence of Wi-Fi or cellular network. An ESP32-based LoRa microcontroller embedded with a special firmware called rf95modem is connected to a smartphone through Bluetooth BLE. On the smartphone, a cross-platform chat application is developed to enable users to type messages. The application's structure is channel based where users can send messages to publicly available channels. The application is then integrated with delay-tolerant network (DTN) software which helps handle extreme situations of delays. Höchst et al. use a simple communication protocol for data transmission where data packets are labeled with sequence numbers. Therefore, if a packet is lost, the receiver is aware and requests for retransmission. Tests were conducted between two devices, a fixed-position laptop, and a mobile device. Additionally, the test was made in urban and rural areas. The maximum distance was 2.89km in the city and 1.64km in a rural area for an SF value of 12. Sciullo et al. (2018) developed an emergency communication system named LOCATE, using a D2D connection with mobile phones. In this study, the LoRa model is connected to a mobile phone using a universal serial bus (USB). The application has options to change LoRa parameters like the spreading factor, coding rate, bandwidth …etc. Based on the testing, the maximum distance reached by the LoRa was 1km. the LOCATE system functions with multi-hop D2D communication between LoRa devices and therefore, does not require using LoRaWAN gateways. The application is designed to support three types of users logically: the emergency source, the one who needs help, the emergency solver, the one who has solutions, and the emergency relay, who cannot help and therefore passes the message to other nodes. Conse-

quently, there are two types of replies, Emergency request, and emergency reply. A request message is retransmitted if not replied to in the specified timeline. In the scenario proposed, there is one fixed source, and the other nodes are mobile. Results show that the request time decreases by increasing the number of nodes that can reply, and by using LoRa as the system showed more time to reply when using Wi-Fi. Results also show that after 30 minutes, almost all emergency sources are replied to.

To conclude, the reviews above showed the superiority of LPWAN technologies, especially the LoRa over other communication technologies by reviewing messaging systems by other technologies and comparing them to systems of the same purpose made by LoRa.

All the provided messaging systems by LoRa that were mentioned above used simple protocols for data packet transmission and lacked features like security, proper data packeting, and retransmission protocols. My project aims to fill these gaps by providing network specifications like security, device addressing, and other needed network parameters.

Chapter 3 Overview of LPWAN and LoRa Technology

This project aims to implement a network with LoRa technology which is a class of LPWA network. Therefore, it is crucial to provide a deep and intuitive understanding of the technology behind LPWAN and LoRa. Hence, this chapter is dedicated to give a detailed explanation of LPWAN, LoRa, and the hardware specifications of the LoRa transceivers module used for this project.

This chapter is divided into three sections. Section 3.1 is about LPWAN and some examples of its technologies. Section 3.2 covers LoRa and its characteristics. Finally, section 3.3 illustrates the hardware and software components used in this project.

3.1 LPWAN

LPWANs have gained popularity as the ideal technology for IoT devices. Even though the term LPWAN was first used in the late 2000s and early 2010s, there were similar network architectures that emerged in the early 1990s. One of the main drives for the rise of LPWAN technology was the increased number of devices connected to the internet that required low-cost and low-data devices.

3.1.1 LPWAN characteristics

A. Long range

Compared with other wireless technologies, LPWAN is a long-range network as it can propagate from a few kilometres to 15 kilometres based on location. (Raza et al., 2017). LPWAN operates below one GHZ band which is lower than other wireless networks like wireless local area networks (WLAN) and Bluetooth which operate on 2.4GHZ. Lower frequencies penetrate better through obstacles, and therefore, signals can travel longer distances.

B. Low power consumption

As the name suggests, Low power is a main feature of the LPWAN. Battery life of devices is expected to last up to ten years with LPWAN connectivity. To fulfill these standards, most LPWANs implement star topology. In star topology, end de-

vices are directly connected to the base station (source node), therefore they do not waste energy on listening to multiple nodes. In contrast to mesh topology which connects multiple nodes and therefore requires high power to deploy.

C. Low cost

LPWANs implement several techniques to maintain low cost for end devices. For instance, LPWANs reduce the hardware complexity of end nodes onto base stations which decreases cost per node. Additionally, most LPWANs use the license-free ISM band and thus do not need to be paid for.

3.1.2 LPWAN network model

The rise of IoT is one of the factors that helped the re-emergence of LPWAN. Even though there is no standard architecture for LPWAN implementation of IoT, a generic four-layered model can be designed. First is the physical layer that configures the physical parts of the end devices (sensors and actuators), and the transmission of data via LPWAN from these devices to the gateways that act as an interface between end devices and the IP network. Then comes the network layer where the data that was received from gateways is sent over to network servers. In the servers, data is processed, analysed, and for each application, and this is called the information processing layer. Finally, the data is sent to customers through the application layer (Chaudhari and Zennaro, 2020). The figure below illustrates the LPWAN architecture plot.

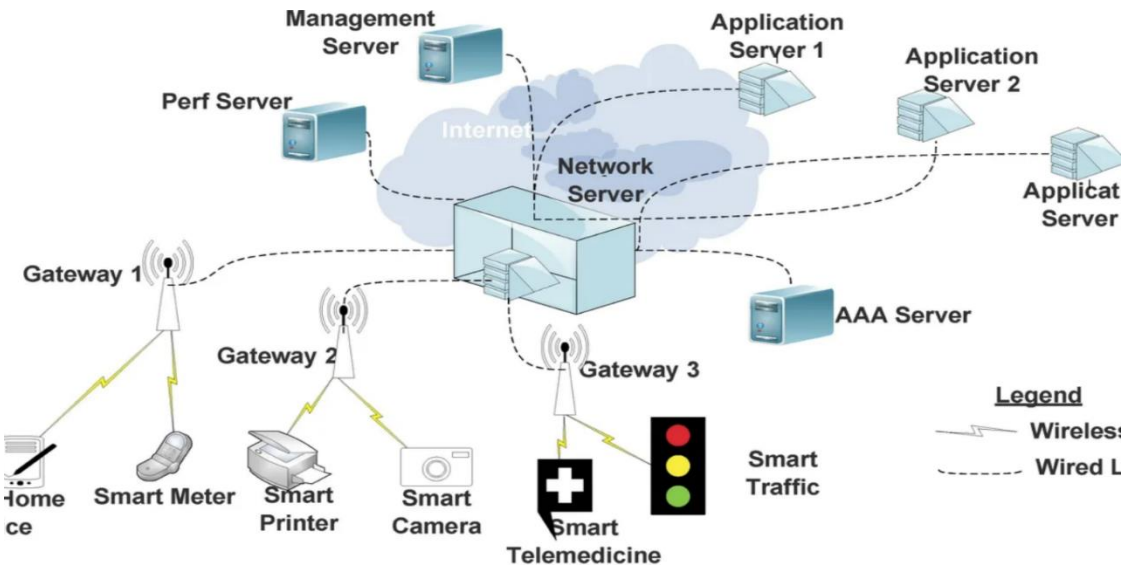


Figure 3.1 LPWAN architecture

3.1.3 LPWAN technology

In this section, a few LPWAN technologies which fall into different categories are highlighted. Generally, LPWANs are non-cellular-based technologies, but some technologies are cellular-based. These types can use licensed or unlicensed frequencies and they can be open standards, meaning they follow public standards and protocols, or they can be proprietary, which means they are technologies developed and owned by a specific company. Some of these technologies are briefly discussed next.

A. Sigfox

Sigfox, founded in 2010, is the first modern LPWAN. Sigfox is a non-cellular, proprietary network that uses ultra-narrowband in the unlicensed ISM bands. This network uses slow modulations like the differential binary phase shift keying (DBPSK) modulation for uplink and Gaussian frequency-shift keying (GFSK) modulation for downlink. Thus, this enables Sigfox to achieve long ranges of 30 to 50 km in rural areas and 3 to 10 km in urban areas. However, this comes at the expense of low data rates of a maximum of 100 bps.

B. LoRa

LoRa is also a proprietary network that operates in the non-cellular unlicensed ISM frequency bands. LoRa the 868 MHz in Europe, 915 MHz in North America, and 433 MHz in Asia ISM bands (Mekki et al., 2018). However, unlike Sigfox, LoRa uses a wideband spread spectrum modulation called the chirp spread spectrum (CSS). An important feature of the CSS is the spreading factor that compromises between data rates and range. Thus, data rates can vary from 300 bps to 50 kbps. The LoRa Alliance has developed the LoRaWAN protocol that serves as a network layer protocol for the LoRa physical layer technology.

C. NB-IoT

Narrowband IoT (NB-IoT) is a cellular LPWAN technology that coexists within the global system for mobile communication (GSM) and long-term evolution (LTE) in licensed frequency bands of 700, 800, and 900 MHz (Chaudhari and Zennaro, 2020). NB-IoT uses single-carrier frequency-division multiple access (SCFDMA) for uplink, and orthogonal frequency-division multiple access (OFDMA) for down-link communication. NB-IoT changes the LTE specifications to create a low-power network with lower data rates which are around 20 kbps for uplink and 200 kbps for downlink communications.

3.2 LoRa

As an LPWAN technology, LoRa is often related to IoT systems for its low-power, long-range abilities, which are ideal for battery-powered IoT systems. LoRa has many advantages over other LPWAN technologies. For example, LoRa operates on unlicensed frequency bands, hence, it is free, unlike NB-IoT and other technol-ogies that operate on licensed frequency bands and therefore require licensing fees. As an unlicensed radio frequency (RF) technology LoRa offers data rates ranging from hundreds of bits per second (bps) to kilobits per second (kbps). In contrast, other unlicensed technologies like Sigfox, do not exceed a few hundred bps. While LoRa is a physical layer technology, LoRa also has a network layer called LoRaWAN, which is a standardized network protocol by *LoRa Alliance.*

LoRa's physical layer is responsible for the RF signal properties that include frequency, modulation technique, coding rate, and other transmission characteristics. The main feature of the LoRa physical layer is the chirp spread spectrum (CSS) modulation technique. As a spread spectrum modulation, CSS spreads the signal over its entire bandwidth. This technique creates a robust and noise-resistant signal because if some frequencies interfere with noise, only the parts of the signal within that frequency channel are affected by noise while other parts of the signal on other frequency channels remain intact.

The basis of the CSS is the chirp waveform which is the carrier wave for the modulation. The chirp is a sinusoid whose frequency linearly increases (up-chirp) or decreases (down-chirp) over time. The chirp waveform is depicted in the frequency and time domain in Figure 3.2. The first step in modulating the chirp is to encode data into the carrier wave, thus creating a symbol. The number of bits used to encode a chirp is determined by the spreading factor *SF*. The spreading factor is a number from 7 to 12 that represents how many bits are used per symbol. The total possible symbols for a spreading factor are represented as 2^{SF}. For example, a spreading factor of 7 will produce 128 different symbols and it doubles whenever the spreading factor is added by one.

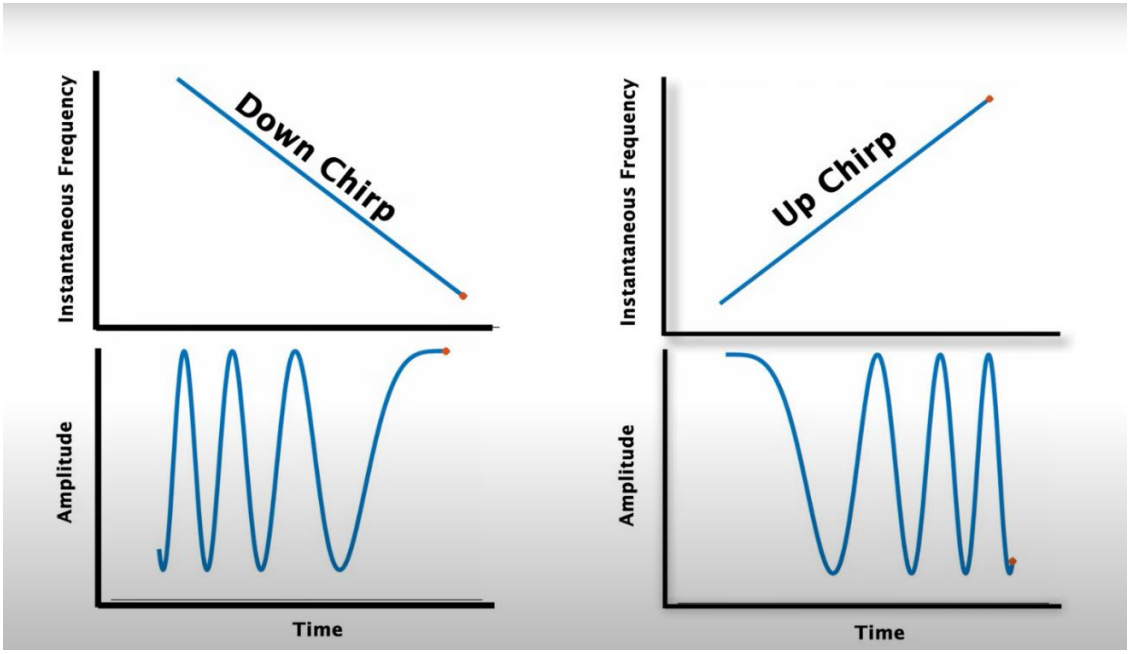


Figure 3.2 The chirp waveform

The spreading factor is also used to divide the chirp wave into 2^{SF} samples per chirp. Lorenzo (2017) provides the mathematical background of a chirp wave modulation in his paper. The formula is shown in the figure below.

$$c(nT_s + kT) = \frac{1}{\sqrt{2^{SF}}} e^{j2\pi \left[(s(nT_s)+k) \bmod 2^{SF}\right] kT \frac{B}{2^{SF}}}$$
$$= \frac{1}{\sqrt{2^{SF}}} e^{j2\pi \left[(s(nT_s)+k) \bmod 2^{SF}\right] \frac{k}{2^{SF}}}$$
$$\text{for } k = 0 \ldots 2^{SF} - 1.$$

Figure 3.3 The modulated waveform formula

T_s is the period of a single symbol, while T is the period of one sample in a symbol, and therefore $T_s = 2^{SF} * T$ (every symbol has 2^{SF} samples). $s(nT_s)$ is the symbol number formed from the data bitstring of SF size. This number ranges from 0 to $2^{SF} - 1$. $s(nT_s)$ value is the starting point and ending point of the modulated wave, and this is what the symbol is, meaning that each of the 2^{SF} symbols has a unique starting point. K is an integer that increments the instantaneous frequency of each sample. Once the value of $s(nT_s) + K$ reaches the maximum frequency of the symbol (2^{SF}), It rolls back to the lowest frequency. This creates a discontinuity in the waveform that is unique for every symbol value because the starting and ending points for each symbol are its own value from the 2^{SF} spectrum. The figure below plots a chirp modulated signal.

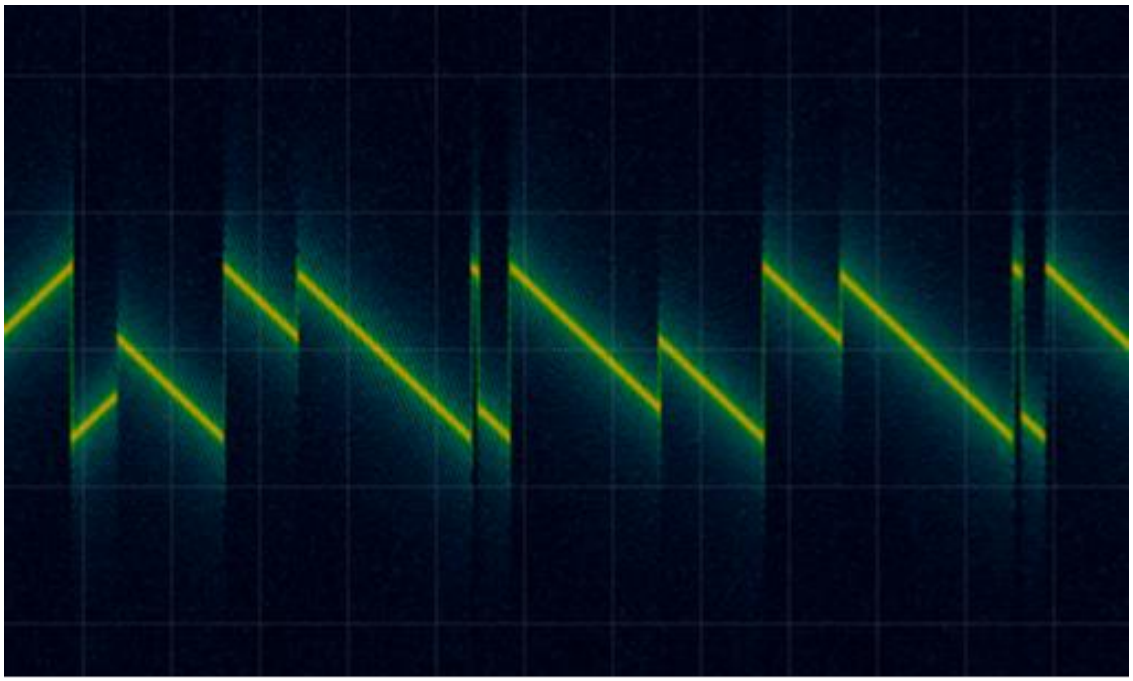


Figure 3.4 Chirp modulated signal

At the receiver, the signal is demodulated. To perform a successful demodulation, the bandwidth and spreading factor for both sender and receiver have to be identical. Also, the received signal has to align with the demodulating window to perform calculations accurately. A bitstring, called the *preamble*, made of 8 up-chirps for synchronization and 2 down-chirps to indicate the start of data symbols is added to the beginning of each packet to signify the start of a data packet (Bizon et al., 2020). Lorenzo (2017) proposes the demodulation of the transmitted signal as the projection of the received signal onto all possible symbols, and choosing the symbol that gets the maximum value from the projection, which is a correlation-based formula to multiply the received signal with each symbol, sample by sample for similarity check. The formula is shown in the figure below.

$$\sum_{k=0}^{2^{SF}-1} r(nT_s + kT) \cdot c^*(nT_s + kT)$$

Received signal Basis symbols

Figure 3.5 Correlation formula

In correlation, we multiply the received signal with the conjugate of the symbols. The symbol with the highest correlation value is the symbol the signal was modulated with. However, this is extremely inefficient due to all the calculations that are made per signal received. For every signal similarity check, $2^{2(SF)}$ calculations are made. Alternatively, Lorenzo (2017) proposes a solution by bringing the basis symbol $c^*(nT_s + kT)$ and applying a mathematical operation that divides it into two terms.

$$= \frac{1}{\sqrt{2^{SF}}} \underbrace{\left(e^{-j2\pi \frac{k^2}{2^{SF}}} \right)}_{\text{Base down chirp}} \underbrace{e^{-j2\pi (s(nT_s)+k \mod 2^{SF}-k) \frac{k}{2^{SF}}}}_{\text{a pure wave at frequency s}}$$

Figure 3.6 Basis symbol derivation

As shown in Figure 3.6, The first term is a base down chirp. Multiplying the received signal $r(nT_s + kT)$ with the down chirp is called *Dechirping*. *Dechirping* removes the chirp modulation component from the received signal, returning the signal to its pre-modulation state. The second term is a collection of sinusoid waves that vary based on the $s(nT_s)$ value. Multiplying the dechirped signal with the second term is equivalent to the *fast Fourier transform* (FFT). In the outcome, you get a set of frequencies that represent the similarity checks with basis symbols where the maximum value is the detected symbol value (starting value). The figure below shows the steps of demodulating the chirp signal.

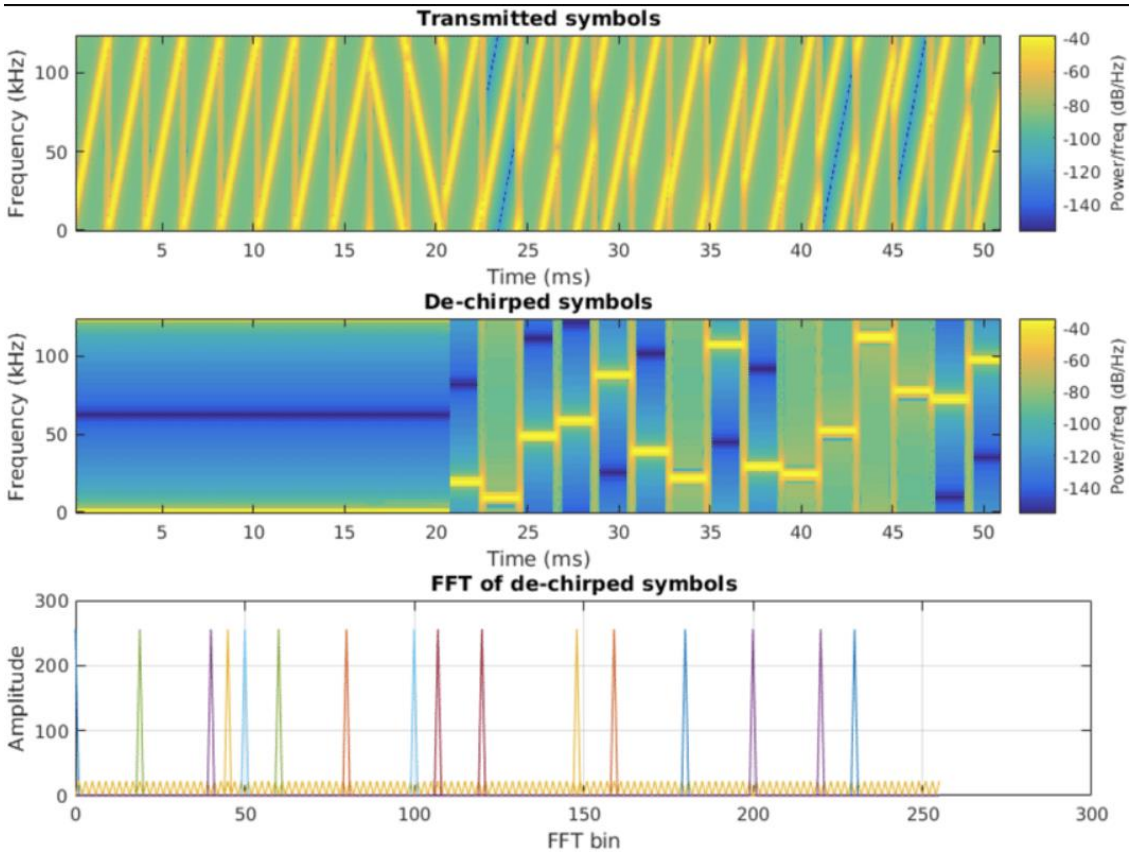


Figure 3.7 CSS demodulation

This unique method of correlating the signal enables the receiver to detect signals 20dB below the noise floor.

In addition to the data, other necessary parameters are transmitted. Figure 3.8 illustrates the LoRa physical layer data packet.

Preamble	PHDR	PHDR_CRC	PHYPayload	CRC

Figure 3.8 LoRa PHY data packet

Data packet parameters are explained as follows:

- **Preamble**, as mentioned before is the chirp sequence used for synchronization.
- **PHDR (physical header)**, contains metadata about the packet. Some of these parameters are:

    - Spreading factor (SF)
    - Payload size
    - CRC information
    - Coding rate
- **PHDR_CRC (Header CRC)**, is an error-detecting method for the header
- **PHYPayload**, is the pure data. The maximum payload size for LoRa is 256 bytes
- **CRC (Cyclic redundancy check)**, is used for detecting errors in the payload

LoRa is mainly evaluated based on parameters as, the spreading factor (SF), coding rate (CR), bandwidth (BW), symbol rate (R_s), and chip rate (R_c) (Chaudhari and Zennaro, 2020). A higher spreading factor allows for data to travel for longer ranges, but with lower data rate, and vice versa. This is because higher spreading factors allow for data to spread over a wider range of bandwidth, making a more robust signal that can travel longer, but at the same time, this creates a delay because the duration for each symbol transmitted increases, and therefore data rate is lower. The coding rate is the rate of bits that carry information to the total number of bits. The addition of redundant data like error correction bits to the transmitted data is called Forward Error Correction (FEC). LoRa has a standard of 4 coding rates: 4/5, 4/6, 4/7, and 4/8. This denotes that for every 4 bits of information, 5,6,7 or, 8 are generated. By increasing the coding rate, we create a more robust system, but also a system with lower data rates (Falanji et al., 2022).

LoRa didn’t have a standardized network layer until LoRaWAN was created by *LoRa Alliance* in 2015. LoRaWAN is the network layer that falls over the LoRa physical layer that has protocols to forward data safely over the network to its destination (Chaudhari and Zennaro, 2020). LoRaWAN implements a *star-of-star* topology where end devices are linked to a network of gateways with LoRa RF communication, and those gateways are connected to a common server with the internet. LoRaWAN has three classes of communication between end devices and base stations. *Class A* has two timed slots to receive a downlink transmission per uplink transmission. This class has the lowest power consumption rates. *Class B* provides random slots at scheduled times in addition to the other two slots. *Class*

C doesn't have a slot, it continuously listens for signals except when it's transmitting. Because LoRaWAN operates in the unlicensed ISM band, it has constraints on the duty cycle and transmission power. For this reason, and because of the limited range covered by LoRaWAN gateways, this project proposes the idea of creating a private LoRa network that can be simply transported anywhere and doesn't have any constraints regarding its transmission rate or power.

3.3 Hardware and software components

This section covers the hardware components used and their features, as well as the software development tools used to program the devices for this project. In addition, other tools are used as well for testing.

3.3.1 Hardware components

A. LoRa TTGO T-Beam v1.1

The LoRa T-beam model is based on the ESP32 microcontroller unit (MCU). The ESP32 is a development board made by *Espressif Systems* that mainly provides Wi-Fi and Bluetooth connections. The ESP32 board consists of a dual-core processor, Wi-Fi chip, Bluetooth chip, 4MB flash memory, and general-purpose input/output (GPIO) pins. The GPIO pins can be used for many purposes such as analog-to-digital conversion (ADC), digital-to-analog conversion (DAC), touch sensors, serial peripheral interface (SPI) communication, Inter-Integrated Circuit (I^2C) communication, and other uses. On top of the ESP32 board, other components are integrated. The LoRa transceiver chip by *Semtech* is integrated into the ESP32 board. The LoRa chip models differ based on the frequency bandwidth they operate on. According to Yefremov (no date), there are two LoRa chips available for the T-beam, the SX1276 which communicates in the 868/915MHz band, and the SX1278 which communicates in the 433MHz band. The allowed ISM frequency bands differ per region. However, there is no specified band for Iraq, hence, this project will use the T-beam version with the SX1276 which operates in the EU863-870 frequency band which is mainly used in the European Union (EU). In addition to the LoRa chip, the T-beam board also comes

with a global positioning system (GPS) module, specifically the *NE0-6M* model. The GPS module provides navigation for the system. The T-beam also comes with an antenna for better signal detection. Lastly, the T-beam has a built-in battery circuit with a built-in *18650* battery holder attached to the back of the board.

3.3.2 Software components

This project uses the PlatformIO, a cross-platform IDE which is an extension in the Visual Studio code development environment that will be used to program the LoRa transceivers. PlatformIO supports Arduino-based board programming like the TTGO LoRa T-beam. For the mobile application development, Flutter is used. Flutter is also a cross-platform software development kit that uses Dart programming language for application development. Lastly, for testing purposes, A draft app is developed in Flutter to send and receive messages with LoRa boards.

Below is a flowchart that explains the primary steps of implementing the proposed project.

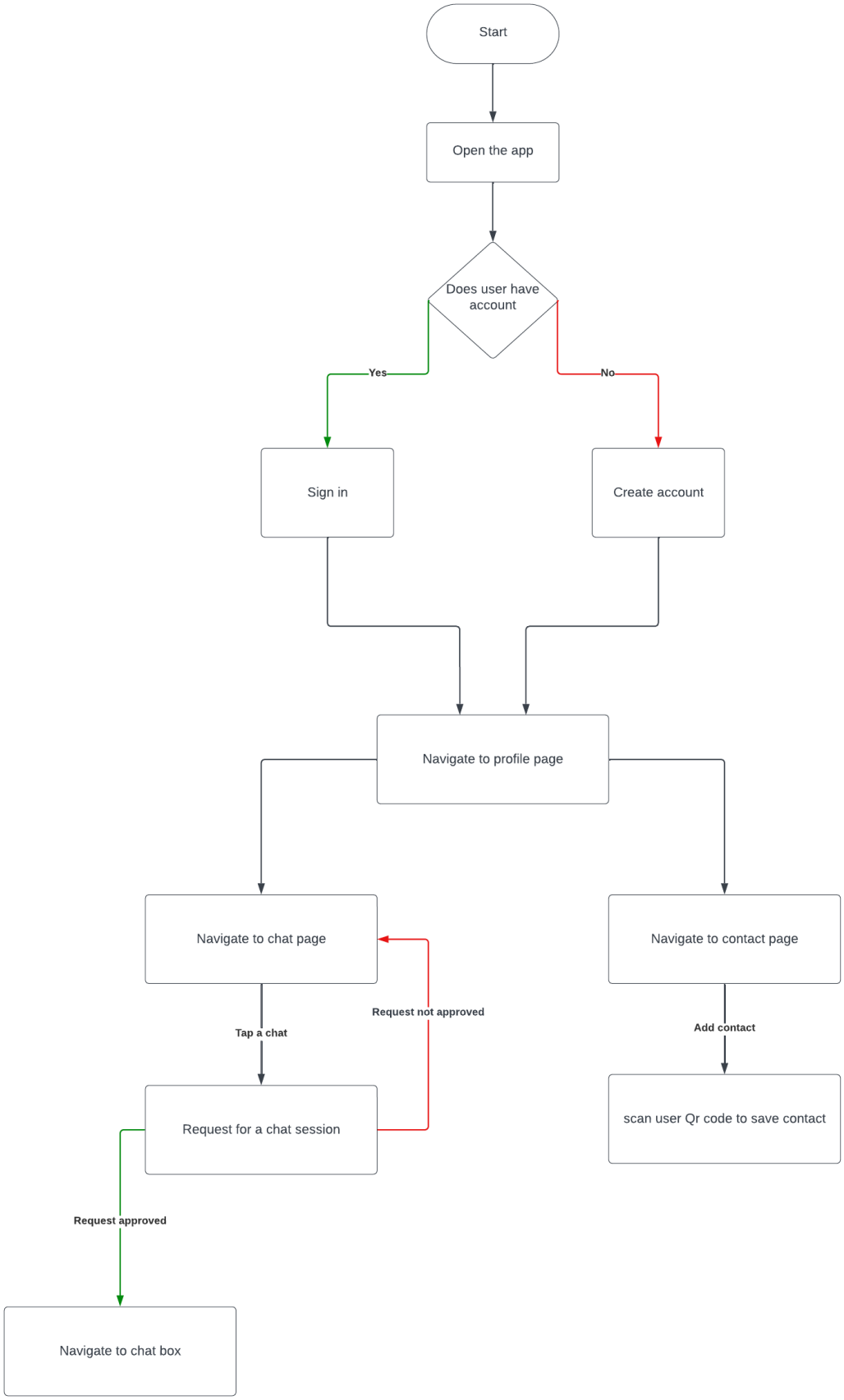


Figure 3.9 System workflow

Chapter 4 Implementation

This chapter dives deep into the details of the implementation phase of this project. The implementation process thoroughly describes the different components of the project, and how they are integrated together to output a final product that is fully functional and usable. Technically, this project touches on a variety of topics such as hardware programming, mobile application development, networking, and security. All these technologies that have been used in this project are divided into sections and explained in detail in this chapter. The hardware part consists of the programming, and service initialization of the LoRa TTGO T-Beam which is the main hardware component used in this project. The TTGO T-Beam model is embedded with LoRa, Bluetooth, Wi-Fi, and GPS chips, however, only LoRa and Bluetooth services are used in this project. On top of the LoRa devices is the networking section where data packets are addressed and defined by creating and attaching headers to them. Subsequently, one-time pad encryption is applied to the data payload. Lastly, a mobile application is developed in Flutter as an intermediary between the user and the network. The backend of the mobile application includes all the functionalities to establish a connection with the LoRa device, while the frontend helps the user interact with the network.

Furthermore, all the elements previously mentioned are implemented and tested and therefore have proven to work. The project successfully establishes the connection between the mobile application and LoRa to send and receive data. Moreover, the LoRa device generates a key that encrypts any data it receives from the mobile application and sends it over to the targeted LoRa device. On the other side, the receiving LoRa device successfully receives data, decrypts it using the same key, and sends it over to the receiver's mobile application to be displayed. The following sections precisely evaluate the coding, algorithms, and configuration that are behind the creation of this project.

4.1 Hardware Programming

In this section, all the details related to configuring the LoRa TTGO T-Beam are explained. All the code for the T-Beam is programmed in C++ using the PlatformIO IDE with Arduino framework in VScode.

4.1.1 LoRa

The LoRa is programmed using a predefined library called LoRa which is used to configure the LoRa components. First and foremost, the parameters of the LoRa like spreading factor, bandwidth, and frequency are initialized as shown in the figure below. It's crucial to note that these parameters should be the same on all LoRa devices communicating together, or else the data will not be transmitted properly.

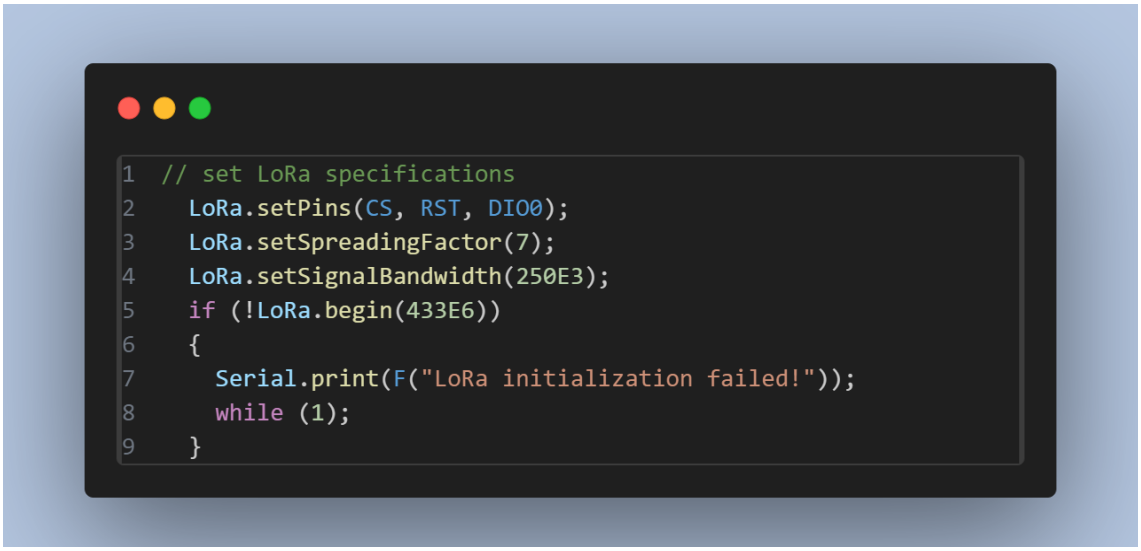
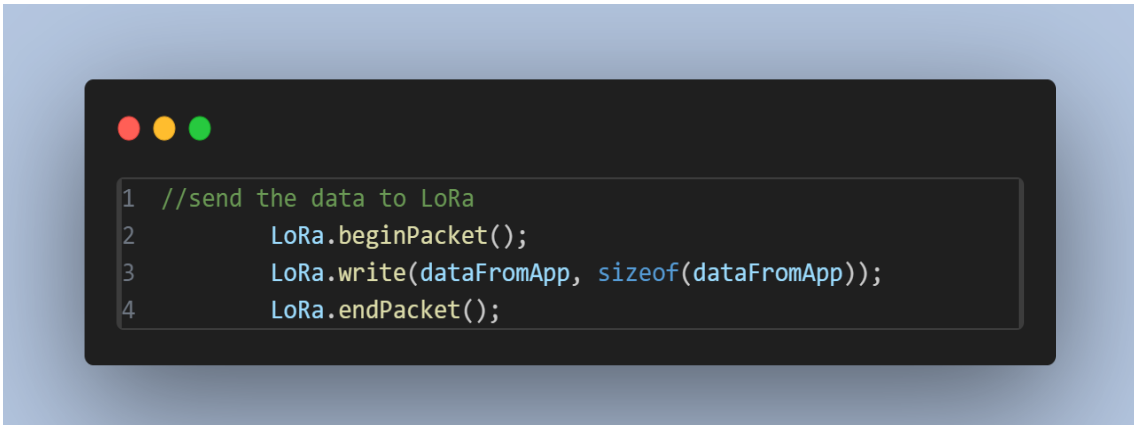


Figure 4.1 LoRa configuration

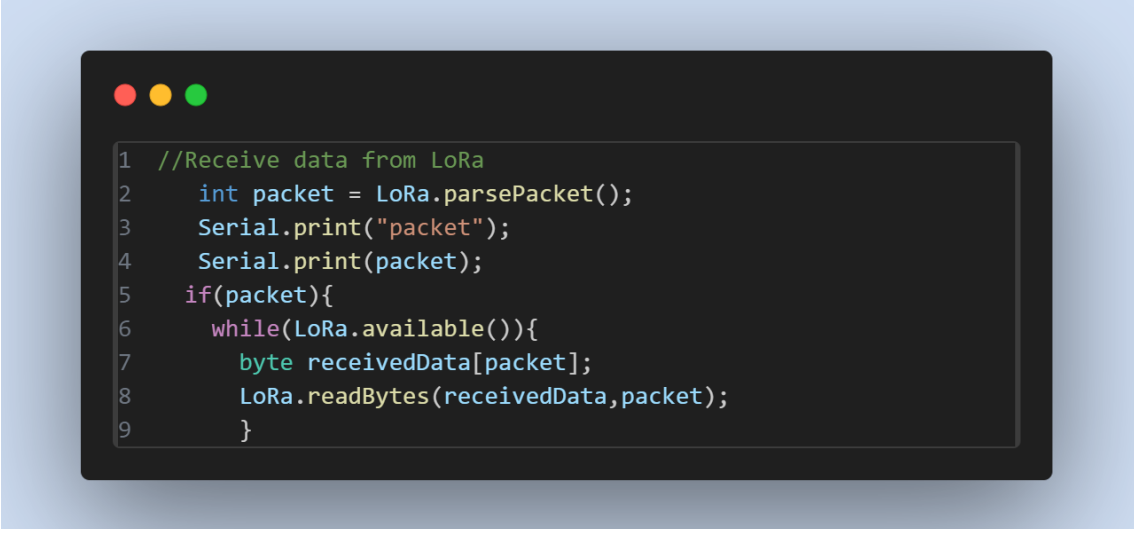
The LoRa provides read and write services. The functions for writing are print() for string datatype and write() for byte datatype. In this project, the write() function is used to access and manipulate data at the byte level. The write() function is used to write the data received from the mobile application to the other LoRa devices. The code is shown in the figure below.



```
//send the data to LoRa
        LoRa.beginPacket();
        LoRa.write(dataFromApp, sizeof(dataFromApp));
        LoRa.endPacket();
```

Figure 4.2 LoRa writing function

For reading incoming data from other LoRa devices, the readBytes() function is used from the LoRa library since the data we expect is in bytes. Reading data is placed in the loop() function to continuously detect incoming data. Data fetched from readBytes() is then sent to the mobile application to be displayed. The code is shown in the figure below.

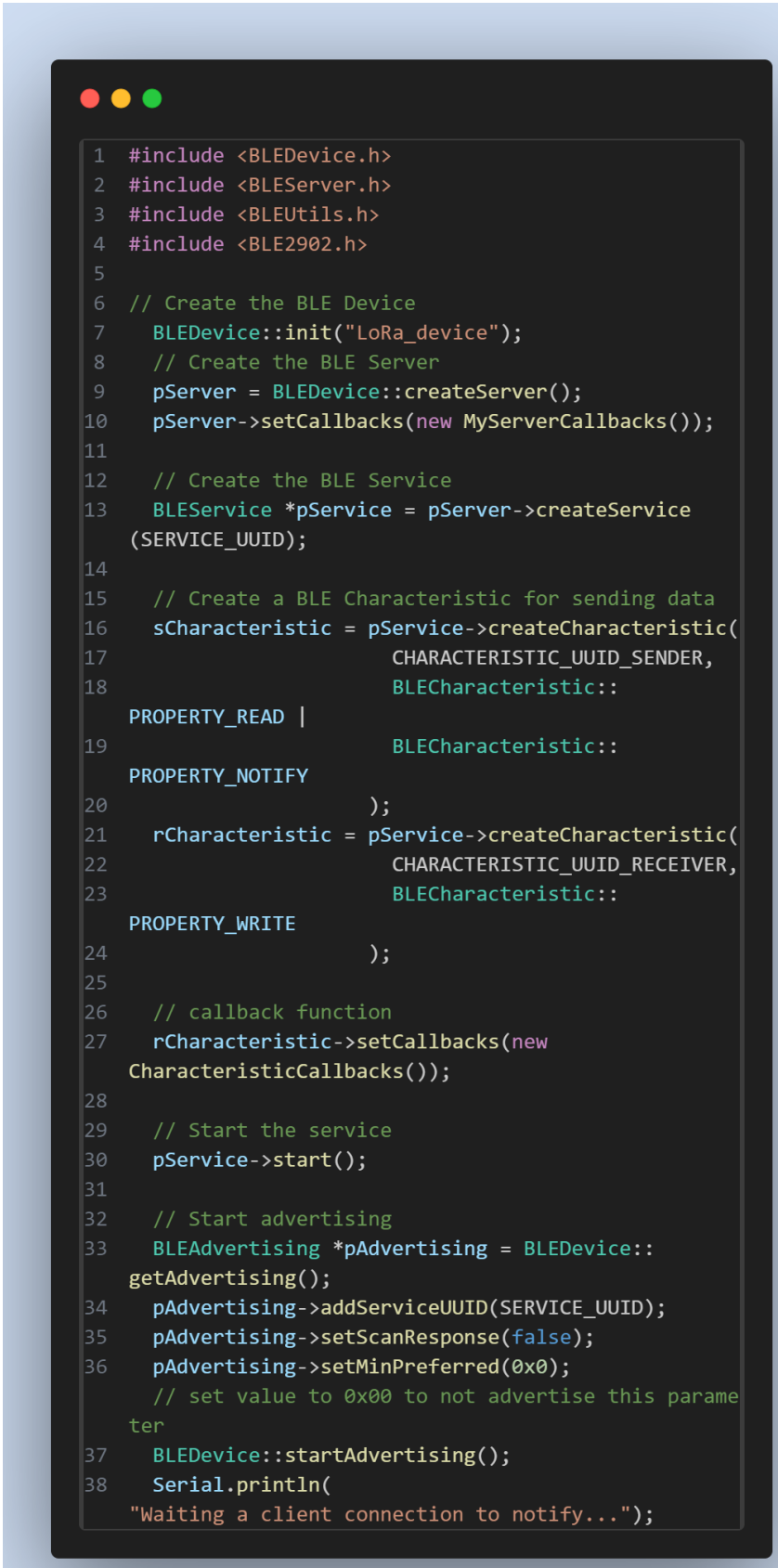


```
//Receive data from LoRa
   int packet = LoRa.parsePacket();
   Serial.print("packet");
   Serial.print(packet);
  if(packet){
    while(LoRa.available()){
      byte receivedData[packet];
      LoRa.readBytes(receivedData,packet);
      }
```

Figure 4.3 LoRa reading function

4.1.2 Bluetooth LE

To configure the Bluetooth LE, a pack of libraries is used to initialize the service, characteristics, and advertising of the BLE as shown in the figure below.



```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

// Create the BLE Device
  BLEDevice::init("LoRa_device");
  // Create the BLE Server
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());

  // Create the BLE Service
  BLEService *pService = pServer->createService
(SERVICE_UUID);

  // Create a BLE Characteristic for sending data
  sCharacteristic = pService->createCharacteristic(
                      CHARACTERISTIC_UUID_SENDER,
                      BLECharacteristic::
PROPERTY_READ |
                      BLECharacteristic::
PROPERTY_NOTIFY
                    );
  rCharacteristic = pService->createCharacteristic(
                      CHARACTERISTIC_UUID_RECEIVER,
                      BLECharacteristic::
PROPERTY_WRITE
                    );

  // callback function
  rCharacteristic->setCallbacks(new
CharacteristicCallbacks());

  // Start the service
  pService->start();

  // Start advertising
  BLEAdvertising *pAdvertising = BLEDevice::
getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(false);
  pAdvertising->setMinPreferred(0x0);
  // set value to 0x00 to not advertise this parame
ter
  BLEDevice::startAdvertising();
  Serial.println(
"Waiting a client connection to notify...");
```

Figure 4.4 BLE configuration

A BLE network is structured in units called services. A service is a data attribute that holds at least one or more characteristics. A characteristic is a data element that holds the actual data and is assigned to properties that define their usage. These properties can be read, write, notify, …etc. Upon the initialization of our BLE device, a service is created that includes two characteristics, sCharacteristic for notifying and enabling data reading when received by the mobile application, and rCharacteristic for enabling incoming data writing to the board when received by the mobile application. Lastly, a BLEAdvertising object is created to advertise for the BLE device to be scanned by other devices. As depicted in the figure above, rCharacteristic is a pointer to a callback function that is used to read data. In this code, the callback function reads data from the mobile application. Below is the structure of the callback function. The details of the callback function data handling are explained in the networking section.

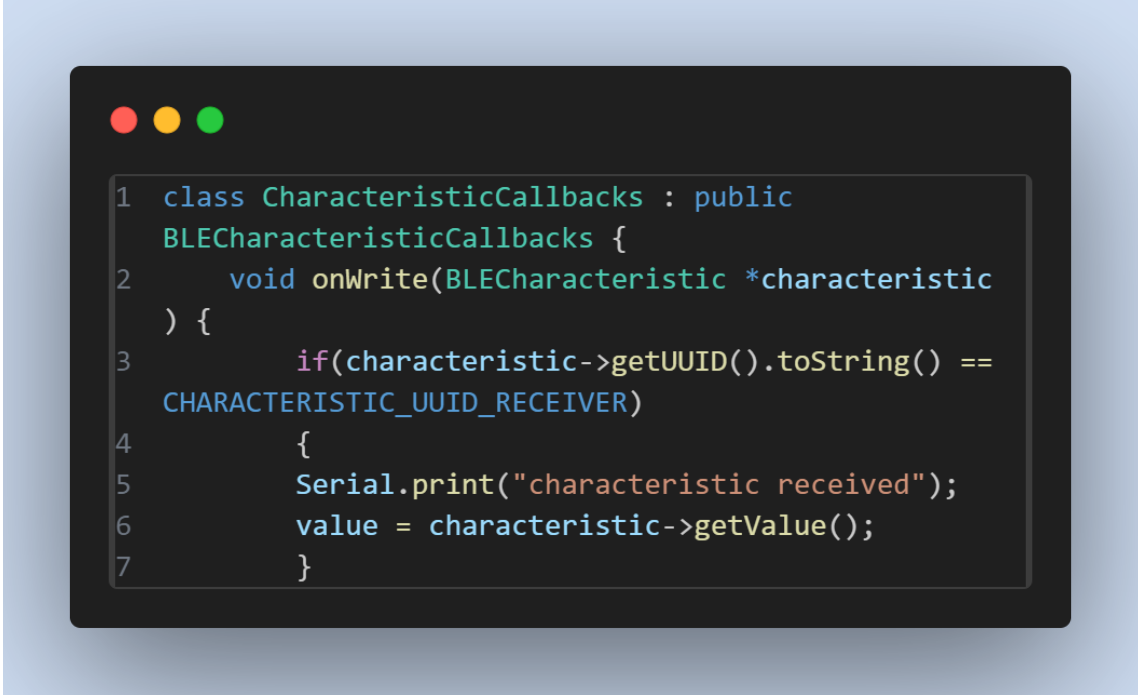


Figure 4.5 BLE Callback function

Additionally, sCharacteristic writes the data received from the LoRa readBytes() function to the mobile application as shown below.

```
1  //Send data to app
2    if (deviceConnected) {
3      sCharacteristic->setValue(receivedData,sizeof(
   receivedData));
4      sCharacteristic->notify(); }
```

Figure 4.6 BLE data writing function

4.2 Networking and Security

4.2.1 Networking

The networking aspect of this project is fundamental to ensure data integrity and security over the network, and this is specifically critical in this project due to using two communication networks: LoRa and BLE. Data packet parameters had to be configured in a way that suited both LoRa and BLE networks. The maximum transmission unit (MTU) for a LoRa packet is 255 bytes while for the BLE it is 512 bytes, and therefore the MTU for this project is restricted to 255 bytes. Moreover, the BLE is firmly secured with encryption algorithms and other security features, while on the other hand, the LoRa is a physical layer network, and therefore is not secured. Provided with that, the one-time pad encryption algorithm is used to encrypt data when transmitted over LoRa alongside a custom key generator method to generate a key for each encryption session. Additionally, Users are authenticated by assigning unique identifiers to each user created on the mobile application. Hence, the data packet is structured into two parts: header and payload. The header includes information related to data identifiers and types, and the payload is primarily the data we want to transmit. The figure below is an illustration of the data packet and how it's organized. Note that there are two types of data packets, each having an MTU of 255, that differ only by the payload which is explained below.

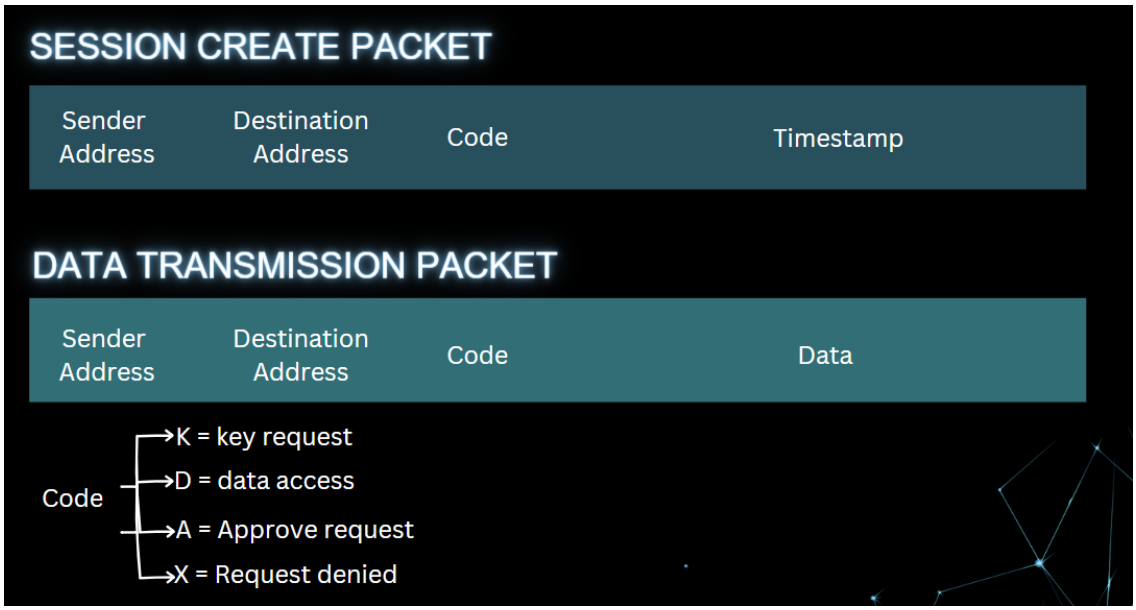


Figure 4.7 Data packet structure

Initially, when we want to request a connection with a user, the encryption key must be shared between both users to create a session with a generated key for encryption and decryption. Consequently, in this case, *the session creation data packet* is used. This data packet's header includes:

- Sender address: is a 10-byte unique identifier that belongs to a user created on the mobile application. This identifier belongs to the sender, thus the initiator of the session.
- Destination address: is also a 10-byte unique identifier that belongs to a user created on the mobile application. However, this identifier belongs to the target user that we want to establish a connection with.
- Code: is a one-byte component that defines what is the type of data being transmitted. In the case of a session request, the code value could be 'k' portraying a request for key exchange, 'a' portraying an approval for the key request, or an 'x' portraying a refusal for the key request.

The payload for this data packet is the timestamp of the sender's device. The current time is fetched from the phone's system clock, and the timestamp is calculated in microseconds using a method in Flutter. The timestamp exchange between both users is necessary to create an identical session encryption key.

Once the timestamps are exchanged and the key is generated, both users are ready to send and receive data. For data transmission, *the data transmission packet* is used. This packet has the same header as the previous one except the code which has the value ‘d’ that depicts that the payload contains data. Subtracting the header from the MTU for our network results in the maximum size for possible data transmission which is 234 bytes.

The entire logic behind the data packets and the transmission is programmed on the TTGO LoRa. Firstly, the data sent from the user’s mobile phone, which is handled in the callback function of the BLE that listens to data from the connected phone is depicted in the figure below. The data received from the mobile application is handled depending on the code character:

- If the code is ‘k’, then the user wants to request a session with the destination address and has attached its timestamp in the payload. In this case, the user’s timestamp is saved in the variable *tsmp1* and the data packet is sent to the destination via LoRa.
- If the code is ‘a’, then the user has approved a key request that was previously sent to them and has attached their own timestamp to send to the sender of the request to create the session key. In this case, the user’s timestamp is saved in the variable *tsmp2* given that the sender of the session has already sent their timestamp and has been saved in the variable *tsmp1*, the generateKey() function is called, and the return value which is the key is saved in the variable *key.*
- If the code is ‘x’, then the user has refused to create a connection with the sender of the key request. In this case, the timestamp that was received from the sender of the key request is nullified and the data packet is sent via LoRa.
- Lastly, if the code is ‘d’, then the user is sending data to a user that they have previously established a session with. In this case, data is encrypted using the encrypt() function, and the return value which is the encrypted data is assigned to the variable *cipher* and sent to the receiver via LoRa, and the callback function is immediately returned.

```
//receive data from the app
class CharacteristicCallbacks : public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *characteristic) {
        if(characteristic->getUUID().toString() ==
CHARACTERISTIC_UUID_RECEIVER)
        {
        value = characteristic->getValue();
        byte dataFromApp[value.length()];
        memcpy(dataFromApp, value.c_str(), value.length());
        //key session
        if(dataFromApp[20] == 'k'){
          Serial.println("k");
          //send your timestamp, and request for other side's timestamp
          for(int i = 0; i < 16; ++i){
            tsmp1[i] = dataFromApp[i + 21];
            }
          }
        //key request approved
        else {
          if(dataFromApp[20] == 'a'){
            Serial.println("a");
          for(int i = 0; i < 16; ++i){
            tsmp2[i] = dataFromApp[i + 21];
            }
        key = generateKey(tsmp1,tsmp2);
          }
          //key request denied
          else {
            if(dataFromApp[20] == 'x'){
              Serial.println("x");
          // tsmp1 --> NULL, device not recognized
          memset(tsmp1, 0, sizeof(tsmp1));
        }
        // data sending
        else if(dataFromApp[20] == 'd'){
          Serial.println("d");
          cipher = encrypt(dataFromApp,sizeof(dataFromApp),key);
          LoRa.beginPacket();
          LoRa.write(&cipher[0], cipher.size());
          LoRa.endPacket();
          return;
        }
        else {
          Serial.println("data not recognized");
        }
          }
        }
        //send the data to LoRa
        LoRa.beginPacket();
        LoRa.write(dataFromApp, sizeof(dataFromApp));
        LoRa.endPacket();
        }
        }
};
```

Figure 4.8 handling data from the mobile application

Likewise, the same logic is applied to data received to the LoRa. The code for receiving data via LoRa is displayed in the figure below. Data is received when the function parsepacket() is not zero, and thus, the data read by the LoRa is handled based on the code byte value, and then sent to the mobile application:

- If the code is ‘k’, this means that a request for a session is sent. The timestamp of the sender is saved in the variable *tsmp1*, and data is sent to the application. The response of the receiver is sent back to the LoRa from the callback function as demonstrated in the previous code.
- If the code is ‘a’, this means that the key request our user has sent to the destination user has been approved. Therefore, the destination user’s timestamp is saved in the variable *tsmp2*, the generateKey() function is called, and its return value is assigned to the *key* variable.
- If the code is ‘x’, this means that the user’s request for a session establishment was rejected, thus the user’s timestamp is nullified, and the received data packet is sent to the mobile application.
- Lastly, if the code received is ‘d’, then this means that data is received from a user that they may or may not have established a connection with previously. The data is decrypted using the *decrypt()* function, and the value is assigned to the *plaintext* variable, and added to the data packet to be sent to the mobile application.

It is important to note that the code programmed in the TTGO device is solely responsible for classifying the data packet elements based on the formulated protocol, to serve data configurations like key generation, encryption, and decryption. Thus, it is not responsible for user authentication. The TTGO receives data either from the BLE or the LoRa, applies the required operations based on the protocol, and then sends it over either with BLE or LoRa while the data authentication is done at the application level. When a data packet is received by the application, it scans if the receiver is found in its database and if it is the intended receiver. If any of these conditions are not met, the application declines the data packet. Equivalently, the application is obliged to attach its identifier and the intended destination’s identifier, then the data packet can be sent to the connected LoRa device.

```
//Receive data from LoRa
  int packet = LoRa.parsePacket();
 if(packet){
   while(LoRa.available()){
     byte receivedData[packet];
     LoRa.readBytes(receivedData,packet);
       //save receiver key
       if(receivedData[20] == 'k'){
         Serial.println("k");
       for(int i = 0; i < 16; ++i){
           tsmp1[i] = receivedData[i + 21];
       }
     }
     // send confirmation to the phone
     else {
     if(receivedData[20] == 'a'){
       Serial.println("a");
       for(int i = 0; i < 16; ++i){
           tsmp2[i] = receivedData[i + 21];
       }
        key = generateKey(tsmp1,tsmp2);
     }
     // request denied
     else {
     if(receivedData[20] == 'x'){
       Serial.println("x");
       // nullify tsmp1
        memset(tsmp1, 0, sizeof(tsmp1));
     }
     //Data received
     else if(receivedData[20] == 'd'){
         Serial.println("d");
         plainText = decrypt(receivedData,sizeof(receivedData),key);
         for(int i = 0; i < plainText.size(); ++i){
           receivedData[i] = plainText[i];
         }
   }
  }
}
  //Send data to app
  if (deviceConnected) {
    sCharacteristic->setValue(receivedData,sizeof(receivedData));
    sCharacteristic->notify();
    delay(300);
// bluetooth stack will go into congestion, if too many packets are sent
    }
    }
  }
```

Figure 4.9 handling data from the LoRa receiver

4.2.2 Security

The security aspect is an essential part of the network developed for this project. The main security feature for this project is data encryption. Applying security on microcontroller-based boards can be a challenge by itself because of the limited data storage and processing power, and this is what limited the options for which encryption algorithm to use. Algorithms like AES and DES are too long and complicated for the limited resources provided by the ESP32-based TTGO LoRa. Eventually, a simple yet secure algorithm had to be implemented to ensure the security and efficiency of the network. In conclusion, the one-time pad is implemented as the encryption algorithm for this project. The one-time pad is known as the unbreakable cipher, but only under strict requirements. Some of these requirements are:

- The key length must be at least as long as the data length. This requirement is fulfilled in this project provided that the key generation method outputs a 256-byte key which is one byte longer than the MTU of the data packet.
- The key must be completely random. This is also resolved because the key is generated from two random values which are the timestamps of both users.
- The key distribution should be very secure. In this project, the key is not shared over the communication channel to start with. Instead, only the parameters of the key which are the timestamps, are shared initially, and the key is generated separately on both sides.
- The key must be used only once. Based on the configuration of the network in this project, key is reused for multiple messages, but not for long because each time the user leaves the chat the session is terminated, and another key request must be sent to create another session with another key.

A user-defined library called *Encrypt* is developed to hold the encryption operations of the LoRa. The library holds three functions: generateKey(), encrypt(), and decrypt(). These functions are demonstrated below.

```
vector<byte> generateKey(byte* tsmp1, byte* tsmp2)
 {
    vector<byte> key_str;
    for(int i = 0; i < 16; ++i){
        rotate(tsmp2, tsmp2 + 15, tsmp2 + 15);
        for(int j = 0; j < 16; ++j){
            key_str.push_back(tsmp1[j+2]^tsmp2[j]);
        }
    }
    return key_str;
}
```

Figure 4.10 Key generation function

The figure above illustrates the function that generates the key for each session. The function takes two parameters: the timestamps for both users. Each timestamp is in microseconds and thus is 16 bytes long. The technique is a nested for loop where the first loop shifts the digits of the second timestamp once to the right, and then the second loop applies the XOR operation of the digits of the first timestamp and the shifted second timestamp. Note that the loop size is 16 to XOR all digits of the timestamps. After the XOR, the first loop shifts the second timestamp once again to the right and then the XOR operation is calculated again. In conclusion, *tsmp2* is right-shifted 16 times, and each time it is XORed with *tsmp1*, and the result for each XOR is pushed into the vector datatype *key_str*. By the end of the calculation, the output *key_str* is a 256-byte truly random key. The figure below better illustrates the key generation method.

Figure 4.11 key generation method illustration

After retrieving the key, that key is applied to data for encryption and decryption. One-time pad is a simple algorithm where in the encryption process, data is XORed with the key byte by byte to output a cipher. Similarly, in the decryption process, the cipher data is XORed with the key, and thus the original data is retrieved. The function for both processes is shown in the figures below. Note that the encryption and decryption are only applied to the payload, whereas the header is not included in this process.

```
vector<byte> encrypt(byte* plaintext,int size, vector<byte> key) {
    vector<byte> ciphertext;
    for (int i = 0; i < size; ++i) {
        if(i <= 20){
            ciphertext.push_back(static_cast<byte>(plaintext[i]));
        }else{
            ciphertext.push_back(static_cast<byte>(plaintext[i]) ^ key[i]
);
        }
    }
    return ciphertext;
}
```

Figure 4.12 One-time pad encryption

```
vector<char> decrypt(byte* ciphertext,int size, vector<byte> key) {
    vector<char> decryptedText;
    for (int i = 0; i < size; ++i) {
        if(i <= 20){
            decryptedText.push_back(static_cast<char>(ciphertext[i]));
        }else{
            decryptedText.push_back(static_cast<char>(ciphertext[i] ^ key[
i]));
        }
    }
    return decryptedText;
}
```

Figure 4.13 One-time pad decryption

4.3 Mobile Application Development

The last section of this project following networking, security, and TTGO board programming is an interface that enables the user to interact with the system created. Therefore, a mobile application is developed in Dart language using Flutter software development kit (SDK) in Android Studio IDE. The advantage of Flutter is that it is cross-platform, meaning the same code developed can run on Android and iOS. This section is divided into two parts: backend and frontend.

4.3.1 Backend Development

The backend of this application is accountable for establishing a connection with the LoRa board via Bluetooth LE. Data sent to the application, as well as the data sent from the application, are analysed based on our network's configurations. In addition, a simple user account system is produced where an account is created for each user to be capable of storing other users' accounts and have theirs shared to communicate. A local storage method by Flutter is used to store all the data related to a user account. The rest of this part thoroughly explains the elements of the backend and the technologies behind them. Throughout the development of this application, a collection of packages was used to provide the necessary dependencies and features it requires. The figure below displays these packages.

```
dependencies:
  flutter:
    sdk: flutter
  flutter_blue: ^0.8.0
  permission_handler: ^11.3.0
  device_info: ^2.0.3
  shared_preferences: ^2.2.3
  qr_flutter: ^4.1.0
  path_provider: ^2.1.3
  iconly: ^1.0.1
  qr_scanner_overlay: ^0.0.2
  mobile_scanner: ^5.1.1
  intl_phone_field: ^3.2.0
  dash_chat_2: ^0.0.21
```

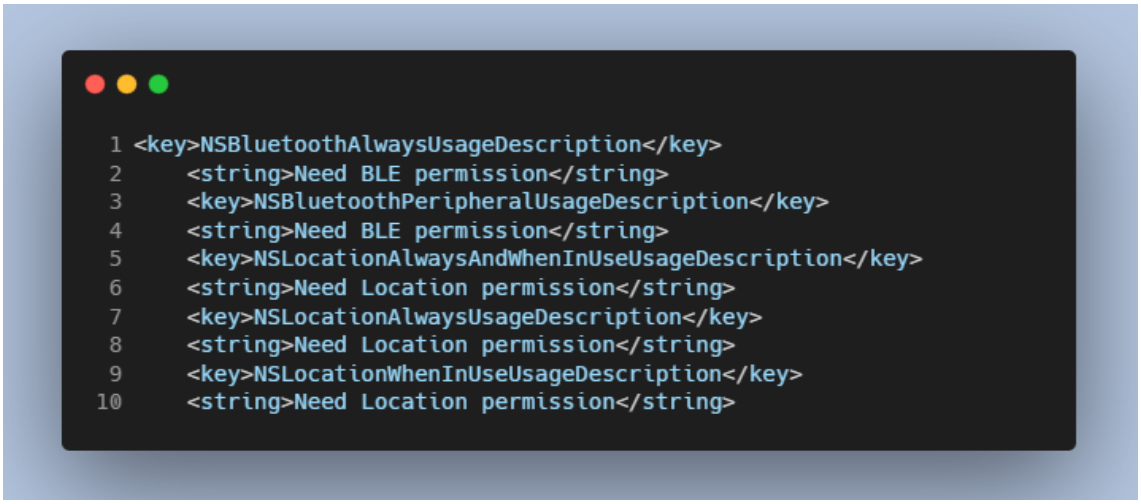
Figure 4.14 Flutter packages

- Flutter_blue: is used for Bluetooth LE communications between BLE devices. This package enables the mobile phone to connect to the TTGO T-Beam.
- Permission_handler: manages the process of requesting permission for sensitive data from Android and iOS. The package is used to request Bluetooth LE permissions from the Android platform.
- Device_info: retrieves device-related information from the phone's system. This package is used to obtain Android platform information when requesting permission.
- Shared_preferences: creates a small local storage space on the user's device. Shared_preferences is crucial to this project because it holds all the data related to the user's account, their personal data, recorded contacts, and chat history.
- Qr_flutter: is a dependency used for generating QR codes. This feature is used to store the user's identifier.
- Path_provider: is used to read and write files. Using this dependency, the QR code is written to a file, and retrieved from there.
- Qr_scanner_overlay: provides a visual frame that helps the user adjust the QR code position when scanning it with the camera.
- Mobile_scanner: this package is used to scan and decode QR codes using the camera.
- Dash_chat2: is an excellent tool that creates chat interfaces. This package provides the outline of a chat UI. This tool is used to develop the chat interface in our application.
- Iconly: package for using Iconly (Icon generator) icons.

These packages are vital to the operation of the application's backend and frontend. The main functionalities provided by these dependencies to the backend are related to Bluetooth LE permission handling and configuration, as well as the generation, scanning, and storing of QR code, and the shared preferences storage management.

A. Bluetooth permissions and configuration

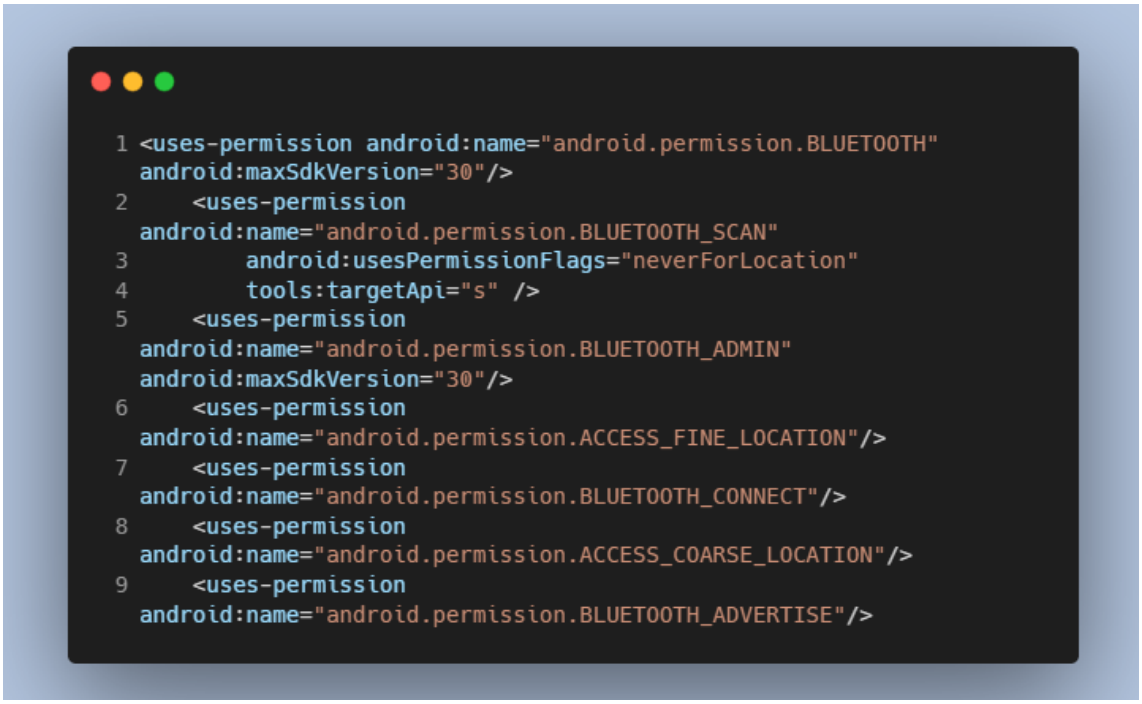
The following permissions should be added to the info.plist file in ios/Runner/Info.plist file.



```
<key>NSBluetoothAlwaysUsageDescription</key>
    <string>Need BLE permission</string>
    <key>NSBluetoothPeripheralUsageDescription</key>
    <string>Need BLE permission</string>
    <key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
    <string>Need Location permission</string>
    <key>NSLocationAlwaysUsageDescription</key>
    <string>Need Location permission</string>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Need Location permission</string>
```

Figure 4.15 ios BLE permissions

In the AndroidManifest.xml in android/app/src/main/AndroidManifest.xml file, the permissions shown in the figure below should be added.



```
<uses-permission android:name="android.permission.BLUETOOTH"
android:maxSdkVersion="30"/>
    <uses-permission
android:name="android.permission.BLUETOOTH_SCAN"
        android:usesPermissionFlags="neverForLocation"
        tools:targetApi="s" />
    <uses-permission
android:name="android.permission.BLUETOOTH_ADMIN"
android:maxSdkVersion="30"/>
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission
android:name="android.permission.BLUETOOTH_CONNECT"/>
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission
android:name="android.permission.BLUETOOTH_ADVERTISE"/>
```

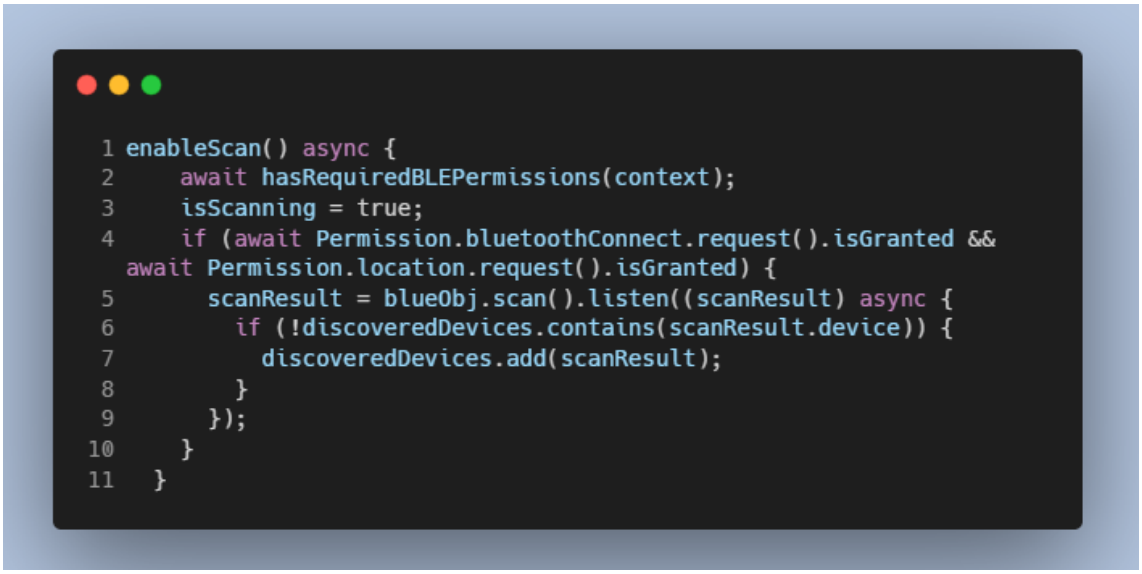
Figure 4.16 Android BLE permissions

Additionally, for Android versions greater than Android 12 some BLE permissions need to be directly handled in the code during runtime. Permission_handler is used to request permission in the code. The code below is used to request those permissions during runtime.

```dart
// Determine whether the current [context] has been granted the relevant [Permission].
  Future<bool> hasPermission(BuildContext context, Permission permission) async {
    var status = await permission.request().isGranted;
    return status;
  }
  // Determine whether the current [context] has been granted the relevant permissions to perform
  // Bluetooth operations depending on the mobile device's Android version.
  Future<bool> hasRequiredBLEPermissions(BuildContext context) async{
    final androidInfo = await DeviceInfoPlugin().androidInfo;
    if(Platform.isAndroid && int.parse(androidInfo.version.release) >= 12){
      return await hasPermission(context, Permission.bluetoothScan) && await hasPermission(context, Permission.bluetoothConnect);
    } else {
      return await hasPermission(context, Permission.locationWhenInUse);
    }
  }
```

Figure 4.17 BLE runtime permission handler

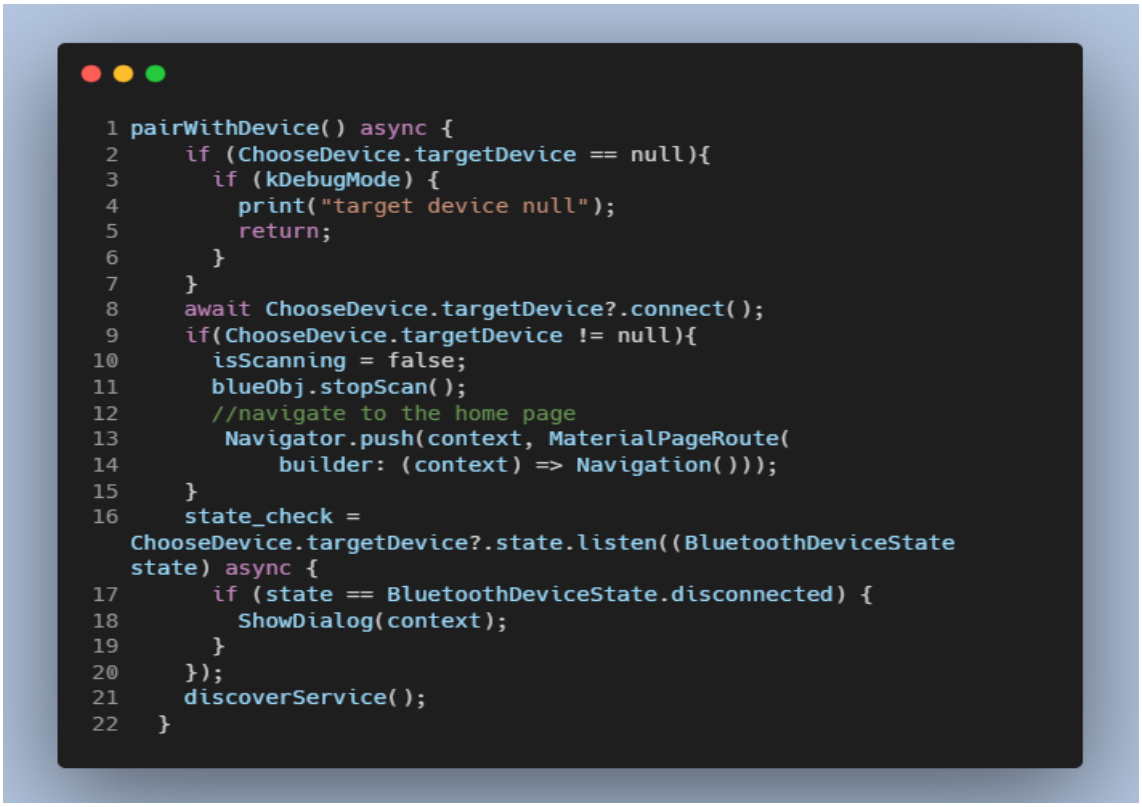
After all the permissions are granted, the device is ready to be paired with other BLE devices and is ready to communicate. Initially, as the permissions are approved, the device enables scanning for BLE devices. When tapping on a device from the scanned devices, the device is paired with our device if permitted to, and scanning for other devices is stopped. Next, services and characteristics for the paired device are scanned and assigned, and devices are ready to send and receive data. This process is mainly provided by three methods in our application. These methods are: enableScan(), pairWithDevice(), and discoverServices(). These methods are depicted, and briefly explained in the figures below.



```
enableScan() async {
    await hasRequiredBLEPermissions(context);
    isScanning = true;
    if (await Permission.bluetoothConnect.request().isGranted &&
await Permission.location.request().isGranted) {
      scanResult = blueObj.scan().listen((scanResult) async {
        if (!discoveredDevices.contains(scanResult.device)) {
          discoveredDevices.add(scanResult);
        }
      });
    }
  }
```

Figure 4.18 BLE enable scan

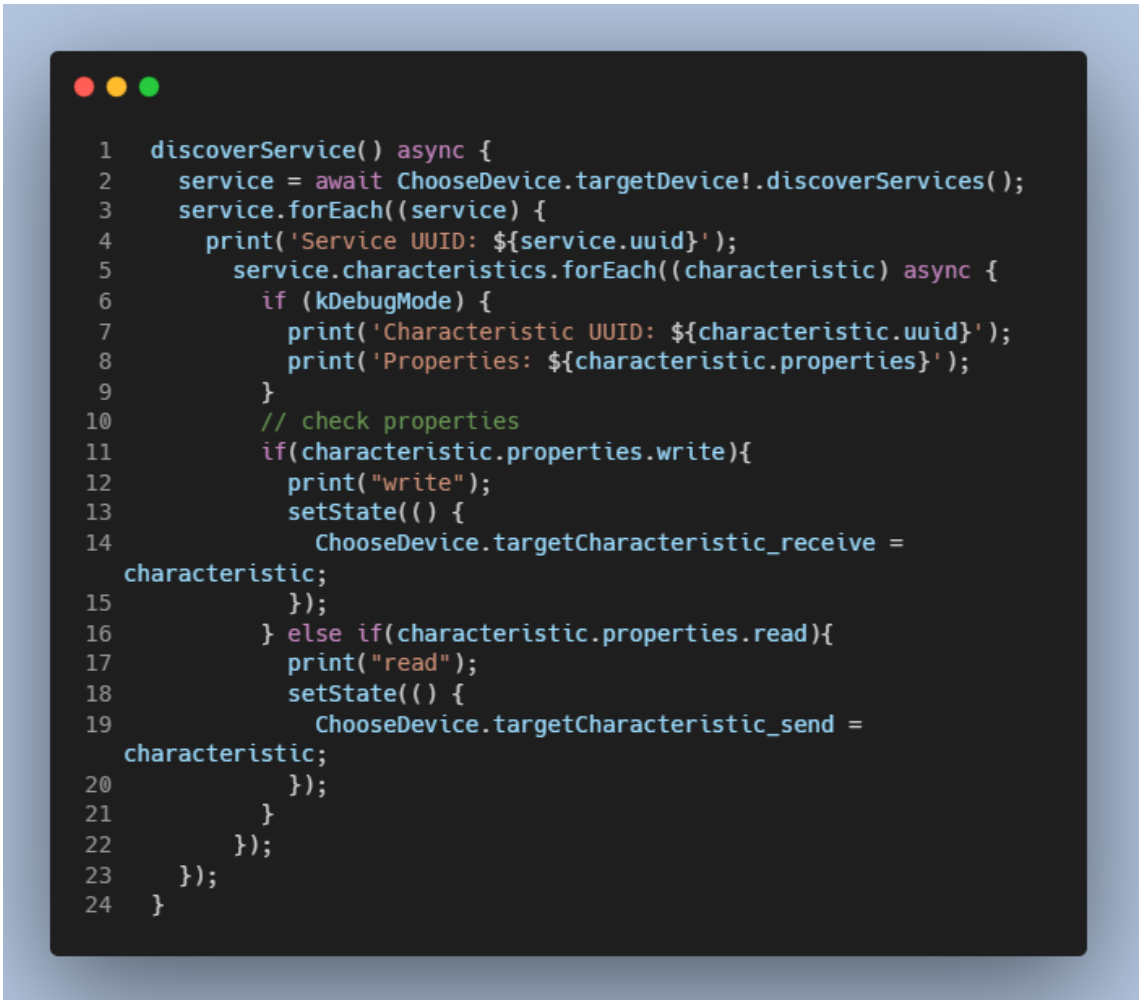
At first, the enableScan() method checks for permission, and if the permissions are granted, scanning for Bluetooth devices is enabled. Note that the scanned device *scanResult.device* is only added to the list of available devices *discoveredDevices* when the device was not added prior to that.



```
pairWithDevice() async {
    if (ChooseDevice.targetDevice == null){
      if (kDebugMode) {
        print("target device null");
        return;
      }
    }
    await ChooseDevice.targetDevice?.connect();
    if(ChooseDevice.targetDevice != null){
      isScanning = false;
      blueObj.stopScan();
      //navigate to the home page
       Navigator.push(context, MaterialPageRoute(
           builder: (context) => Navigation()));
    }
    state_check =
ChooseDevice.targetDevice?.state.listen((BluetoothDeviceState
state) async {
      if (state == BluetoothDeviceState.disconnected) {
        ShowDialog(context);
      }
    });
    discoverService();
  }
```

Figure 4.19 BLE pair with device

Tapping on a device that is in the list of discovered devices, activates the pairWithDevice() method. In this method, if the target device is null, the function is immediately returned, If not the device calls the connect() method, and if the connection is successful, the scanning is stopped immediately, and the discoverService() is called.



```
discoverService() async {
  service = await ChooseDevice.targetDevice!.discoverServices();
  service.forEach((service) {
    print('Service UUID: ${service.uuid}');
      service.characteristics.forEach((characteristic) async {
        if (kDebugMode) {
          print('Characteristic UUID: ${characteristic.uuid}');
          print('Properties: ${characteristic.properties}');
        }
        // check properties
        if(characteristic.properties.write){
          print("write");
          setState(() {
            ChooseDevice.targetCharacteristic_receive =
characteristic;
          });
        } else if(characteristic.properties.read){
          print("read");
          setState(() {
            ChooseDevice.targetCharacteristic_send =
characteristic;
          });
        }
      });
  });
}
```

Figure 4.20 BLE discover services

This method loops through the services of the connected BLE device and scans their characteristics. For our target device, which is the TTGO T-Beam, we have assigned one service to our BLE service which has two characteristics, one with WRITE property and the other with READ property. Therefore, in discoverService() in our application when we scan for the service, we look for the write and read characteristics. The Write characteristic is assigned to *targetCharacteristic_receive*, and the read characteristic is assigned to *targetCharacteristic_send*

Once the devices are successfully paired, the user can navigate to the main page of the application. Furthermore, we can navigate to the chatting page to make sessions with the other users and chat with them. In our application, when you tap on a user's chat to start a chat with them, a request to them at first, and as soon as the request is approved (the keys are successfully generated), the user proceeds to the chat screen and starts chatting. However, if the request is not approved, the chat screen will not open. These functionalities are granted using some methods in our dart code. The logic in these methods corresponds to the logic of the network we created on the TTGO T-Beam device.

The method displayed in the figure below is called when writing data to the BLE for session creation. At first, the method adds the sender's and receiver's identifiers and the code byte to the data packet. Then the payload is added according to the code byte value:

- When the code value is 'k', this indicates that the sender has requested a session by tapping on a chat and wants to open the chat with them. In this case, the sender adds their timestamp to the payload, and a request is sent via BLE.

- When the code value is 'a', this indicates that a request has been sent to our user, and the user has approved the request, and thus, the user sends their timestamp to the TTGO BLE, and to the sender of the request to create the session key.

- When the code value is 'x', this means that a session request was sent to our user, but the user declined the request, and therefore, the user will send the code 'x' with no payload attached to indicate that the session is declined.

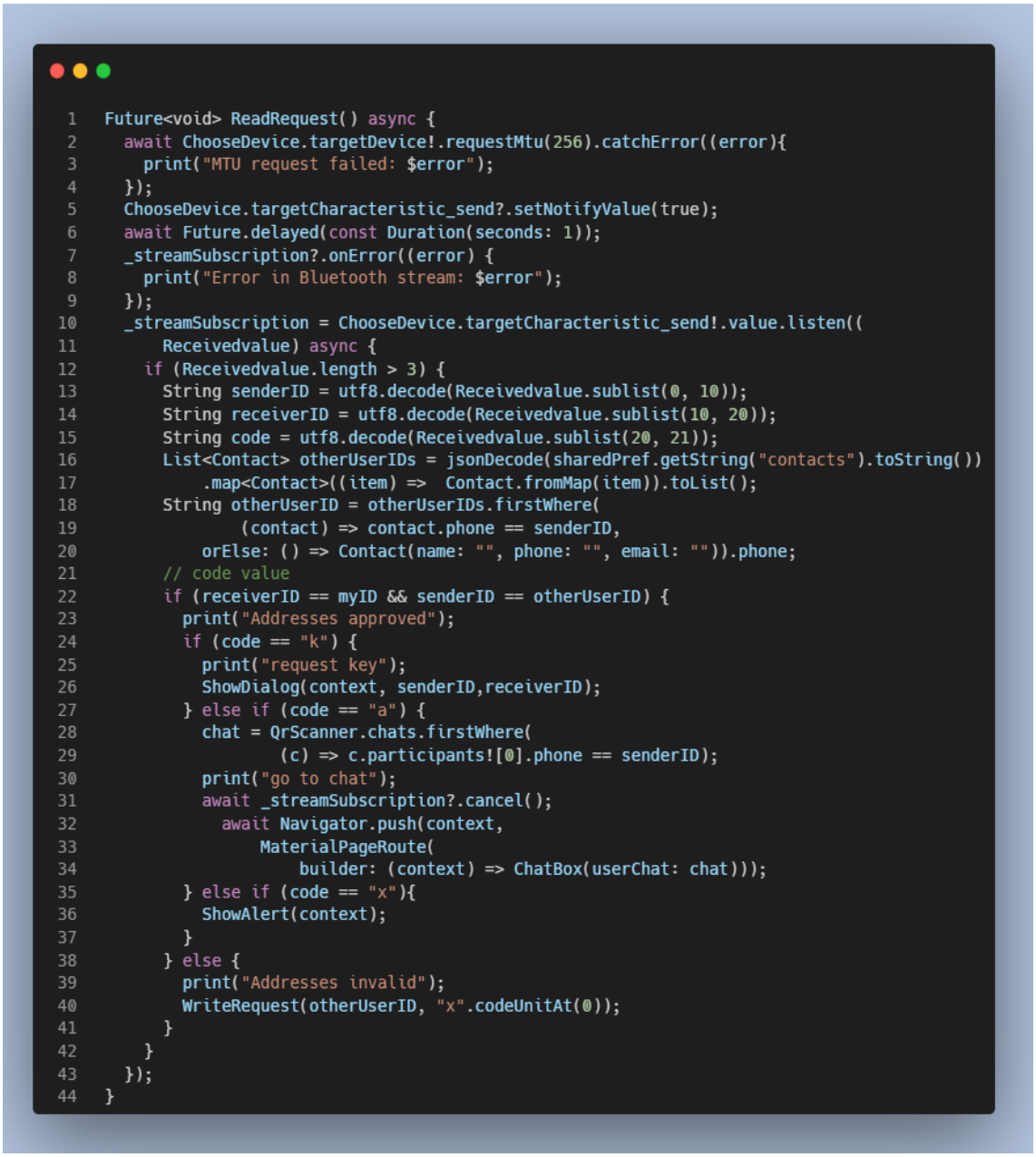
It's necessary to restate that the data is sent via BLE from the application to the TTGO device where data is encrypted and then sent to the receiver's TTGO device, and eventually to their application screen.

```dart
Future<void> WriteRequest(String otherUserID, int code) async {
  List<int> dataPacket = [];
  //first element == sender ID (750 895 6832) => 10 bytes
  for (int i = 0; i < myID.length; ++i) {
    dataPacket.insert(i, utf8.encode(myID)[i]);
  }
  // second element == receiver ID (*** *** ****) => 10 bytes
  for (int i = 0; i < otherUserID.length; ++i) {
    dataPacket.insert(i + myID.length, utf8.encode(otherUserID)[i]);
  }
  // third element == code => 1 byte
  dataPacket.insert(myID.length + otherUserID.length, code);

  if (code == "k".codeUnitAt(0)) {
    print("request key");
    DateTime time = DateTime.now();
    print("unix time");
    String timestamp = time.microsecondsSinceEpoch.toString();
    print(timestamp);
    for (int i = 0; i < timestamp.length; ++i) {
      dataPacket.insert(myID.length + otherUserID.length + 1 + i,
          utf8.encode(timestamp)[i]);
    }
  } else if(code == "a".codeUnitAt(0)){
      print("approve key");
      DateTime time = DateTime.now();
      print("unix time");
      String timestamp = time.microsecondsSinceEpoch.toString();
      print(timestamp);
      for (int i = 0; i < timestamp.length; ++i) {
        dataPacket.insert(myID.length + otherUserID.length + 1 + i,
            utf8.encode(timestamp)[i]);
      }
  } // else if x is the code a denial will be sent
  print("data packet");
  print(dataPacket.length);
  for (int i = 0; i < dataPacket.length; ++i) {
    print(dataPacket[i]);
  }
  await ChooseDevice.targetCharacteristic_receive!.write(dataPacket);
}
```

Figure 2.21 Write data to BLE for session creation method

Likewise, we have a method that reads data from the BLE for session creation. The figure below depicts the method used for reading the data.



```
Future<void> ReadRequest() async {
  await ChooseDevice.targetDevice!.requestMtu(256).catchError((error){
    print("MTU request failed: $error");
  });
  ChooseDevice.targetCharacteristic_send?.setNotifyValue(true);
  await Future.delayed(const Duration(seconds: 1));
  _streamSubscription?.onError((error) {
    print("Error in Bluetooth stream: $error");
  });
  _streamSubscription = ChooseDevice.targetCharacteristic_send!.value.listen((
      Receivedvalue) async {
    if (Receivedvalue.length > 3) {
      String senderID = utf8.decode(Receivedvalue.sublist(0, 10));
      String receiverID = utf8.decode(Receivedvalue.sublist(10, 20));
      String code = utf8.decode(Receivedvalue.sublist(20, 21));
      List<Contact> otherUserIDs = jsonDecode(sharedPref.getString("contacts").toString())
          .map<Contact>((item) =>  Contact.fromMap(item)).toList();
      String otherUserID = otherUserIDs.firstWhere(
              (contact) => contact.phone == senderID,
          orElse: () => Contact(name: "", phone: "", email: "")).phone;
      // code value
      if (receiverID == myID && senderID == otherUserID) {
        print("Addresses approved");
        if (code == "k") {
          print("request key");
          ShowDialog(context, senderID,receiverID);
        } else if (code == "a") {
          chat = QrScanner.chats.firstWhere(
                  (c) => c.participants![0].phone == senderID);
          print("go to chat");
          await _streamSubscription?.cancel();
            await Navigator.push(context,
                MaterialPageRoute(
                    builder: (context) => ChatBox(userChat: chat)));
        } else if (code == "x"){
          ShowAlert(context);
        }
      } else {
        print("Addresses invalid");
        WriteRequest(otherUserID, "x".codeUnitAt(0));
      }
    }
  });
}
```

Figure 2.22 Reading data from BLE for session creation method

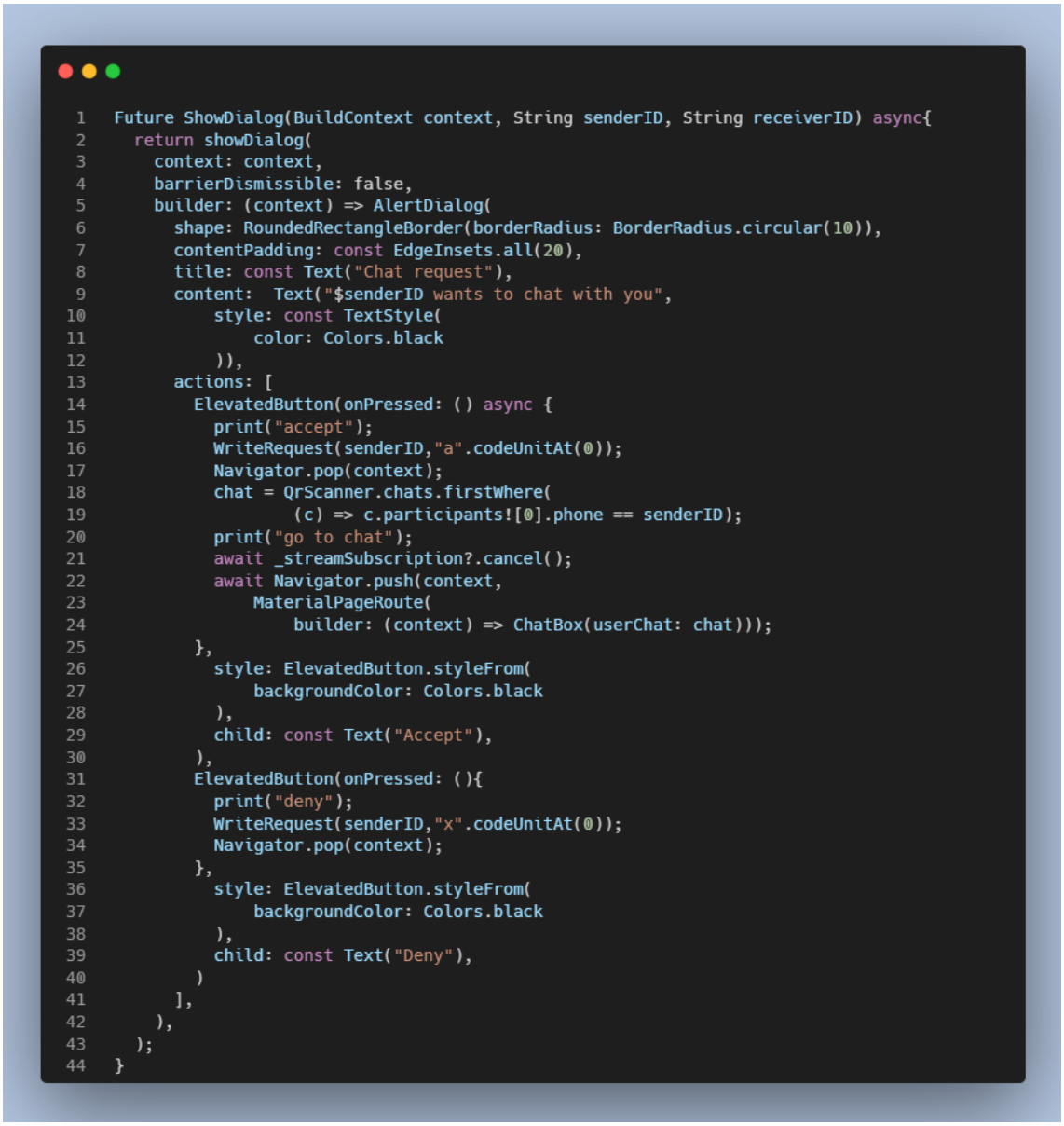
This method is called at the creation of the page. It listens for incoming data packets from the device we're connected to. When a data packet is received, the identifiers of the sender and receiver are checked for validity. If the intended receiver's identifier does not match our user's identifier or the sender's identifier doesn't match any of the contacts in our account, the data packet is dropped, and a data packet with code 'x' depicting data rejection is sent to the sender of the data packet. However, if the sender and receiver identifiers' matches are found, the code byte value is evaluated:

- If the code is ‘a’, this means a session request that we had sent previously has been approved. Therefore, the user is navigated to the chat screen to proceed with chatting.

- If the code is ‘x’, this means a session request that we had sent previously was rejected, and therefore an alert message is displayed, and the user is not allowed to navigate to the chat.

- If the code is ‘k’, this means that a session is requested, and the user can either accept or decline this request. For this, the *ShowDialog* method is called. In this method, if the user presses the *accept* button, the WriteRequest method is called with code ‘a’ which sends a data packet with the code ‘a’ and the user’s timestamp to the sender of the request, and the user is navigated to the chat screen. However, if the user presses the *deny* button, the WriteRequest method is called with code ‘x’ and a rejection data packet with an empty payload is sent to the sender of the request, and the user is not navigated to the chat screen. The *ShowDialog* method is shown in the figure below.

It's important to notice that the stream that listens to incoming data from BLE in the ReadRequest() method is cancelled when the user is navigated to the chat screen.



```
Future ShowDialog(BuildContext context, String senderID, String receiverID) async{
  return showDialog(
    context: context,
    barrierDismissible: false,
    builder: (context) => AlertDialog(
      shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
      contentPadding: const EdgeInsets.all(20),
      title: const Text("Chat request"),
      content:  Text("$senderID wants to chat with you",
          style: const TextStyle(
              color: Colors.black
          )),
      actions: [
        ElevatedButton(onPressed: () async {
          print("accept");
          WriteRequest(senderID,"a".codeUnitAt(0));
          Navigator.pop(context);
          chat = QrScanner.chats.firstWhere(
                  (c) => c.participants![0].phone == senderID);
          print("go to chat");
          await _streamSubscription?.cancel();
          await Navigator.push(context,
              MaterialPageRoute(
                  builder: (context) => ChatBox(userChat: chat)));
        },
          style: ElevatedButton.styleFrom(
              backgroundColor: Colors.black
          ),
          child: const Text("Accept"),
        ),
        ElevatedButton(onPressed: (){
          print("deny");
          WriteRequest(senderID,"x".codeUnitAt(0));
          Navigator.pop(context);
        },
          style: ElevatedButton.styleFrom(
              backgroundColor: Colors.black
          ),
          child: const Text("Deny"),
        )
      ],
    ),
  );
}
```

Figure 4.23 ShowDialog method

When a session is created, and the user is navigated to the chat screen, the user is able to send text messages and receive them. Two methods are created in the chat screen code to read incoming messages from BLE and write messages to the BLE. It's crucial to know that for these two methods, the only code value written and received is 'd' which indicates message data. The figures below display these methods.

```dart
Future<void> writeData(ChatMessage chatMessage,int code) async {
  print("send data");
  List<int> dataPacket = [];
  //first element == sender ID (750 895 6832) => 10 bytes
  for (int i = 0; i < me!.id.length; ++i) {
    dataPacket.insert(i, utf8.encode(me!.id)[i]);
  }
  // second element == receiver ID (*** *** ****) => 10 bytes
  for (int i = 0; i < otherUser!.id.length; ++i) {
    dataPacket.insert(i + me!.id.length, utf8.encode(otherUser!.id)[i]);
  }
  // third element == code => 1 byte
  dataPacket.insert(me!.id.length + otherUser!.id.length, code);
  print("incoming data");
  Message message = Message(senderID: me!.id,
      content: chatMessage.text,
      sentAt: DateTime.now());
  await saveChatMessage(message);
  for (int i = 0; i < chatMessage.text.length; ++i) {
    dataPacket.insert(me!.id.length + otherUser!.id.length + 1 + i,
        utf8.encode(chatMessage.text)[i]);
  }
  print("data packet");
  for (int i = 0; i < dataPacket.length; ++i) {
    print(dataPacket[i]);
  }
  await ChooseDevice.targetCharacteristic_receive!.write(dataPacket);
}
```

Figure 4.24 writeData method

```dart
Future<void> readData() async {
  String senderID, receiverID, code;
  await ChooseDevice.targetCharacteristic_send!.setNotifyValue(true);
  await Future.delayed(Duration(seconds: 1));
  _streamSubscription = ChooseDevice.targetCharacteristic_send!.value.listen((
      Receivedvalue) async {
    print("received data");
    print(Receivedvalue);
    if (Receivedvalue.length > 3) {
      senderID = utf8.decode(Receivedvalue.sublist(0, 10));
      receiverID = utf8.decode(Receivedvalue.sublist(10, 20));
      code = utf8.decode(Receivedvalue.sublist(20, 21));
      if (receiverID == me!.id && senderID == otherUser!.id) {
        if (code == "d") {
          //print data in the UI
          String receivedMessage = utf8.decode(
              Receivedvalue.sublist(21));
          Message message = Message(senderID: senderID,
              content: receivedMessage,
              sentAt: DateTime.now());
          saveChatMessage(message);
        }
      }
    }
  });
}
```

Figure 4.25 readData method

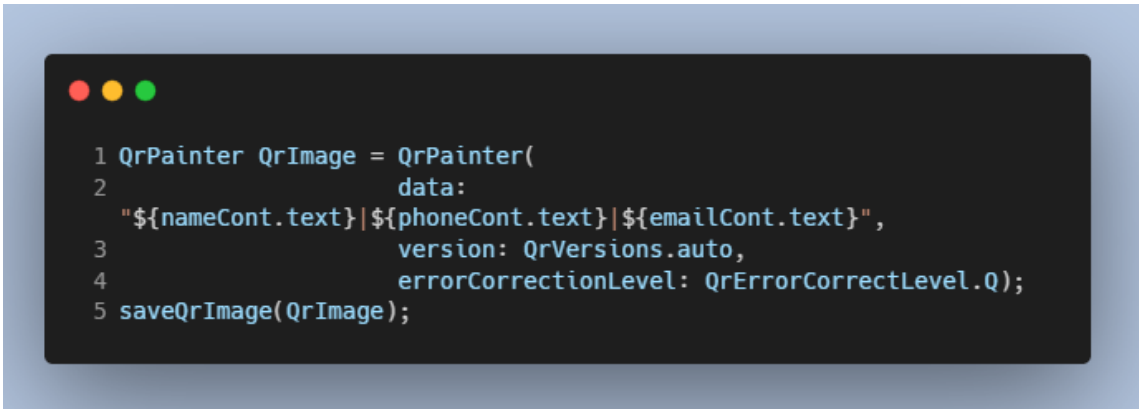
The method writeData() is called with code value ‘d’ when the send button of the message text field is pressed. This method works similarly to the *WriteRequest* method mentioned above, it creates a data packet with the sender’s and receiver’s identifiers, code byte, and the message instance’s content parameter which is the text of the message. The data packet is then sent over BLE.

Moreover, the method readData() reads the incoming data packets, examines the identifiers of the received packet, and once authenticated, checks if the data in the packet is a text message by checking if the code value is ‘d’. If yes, an instance of Message is created for the received message, and it’s saved to the list of messages in the chat instance of the sender and receiver. If not, the packet is dropped.

B. QR code configuration

In our application, a user must have an account to access and use our application. Each account holds three main data entities: the personal information of the users themselves, a list of contacts which are the accounts of the other users that are registered on our user's account, and a list of chats where for each user registered in the contact list, a chat entity is created that they can send and receive messages and all the messages are saved in the database. Thus, a user can only communicate with the users registered on their account. This project uses QR codes to execute this notion. Initially, when a user is created, their personal information is encoded in a QR code and saved into a file. The QR code is then displayed on the user’s profile screen. Consequently, users can scan the QR codes for other users using a QR code scanner that is programmed in the application. Correspondingly, a user is able to save other users’ accounts and likewise display their own accounts to other users to be saved.

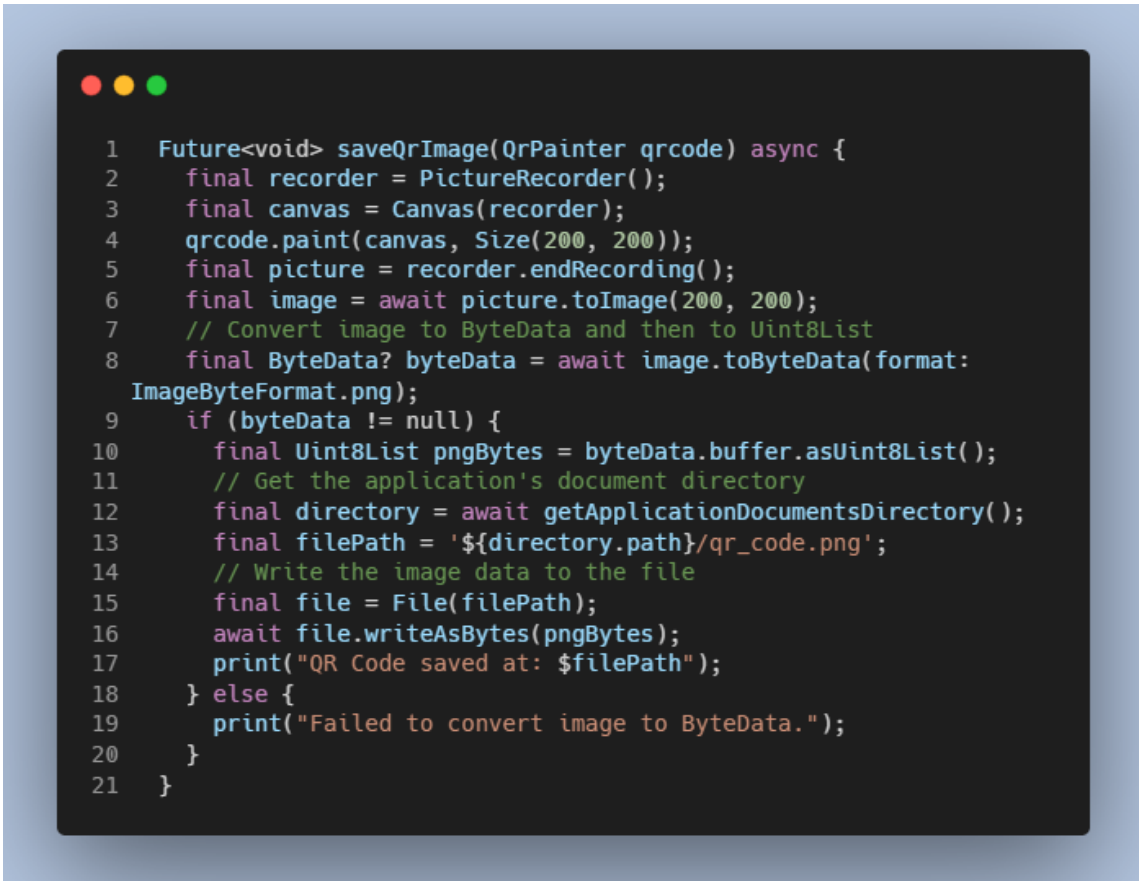
Initially, when the user is signed up, a QR code is created for them, and straightaway moved into a file. The figure below depicts this process.



```
QrPainter QrImage = QrPainter(
                    data:
  "${nameCont.text}|${phoneCont.text}|${emailCont.text}",
                    version: QrVersions.auto,
                    errorCorrectionLevel: QrErrorCorrectLevel.Q);
saveQrImage(QrImage);
```

Figure 4.26 QR code generation

The QR code holds the user's personal data like name, phone number, and email address. After that, saveQrImage(QrImage) is called to save the QR code to a file. In the function, the QR code is converted to uint8List datatype and then it's written to the file in bytes. The function's body is shown in the figure below.



```
Future<void> saveQrImage(QrPainter qrcode) async {
  final recorder = PictureRecorder();
  final canvas = Canvas(recorder);
  qrcode.paint(canvas, Size(200, 200));
  final picture = recorder.endRecording();
  final image = await picture.toImage(200, 200);
  // Convert image to ByteData and then to Uint8List
  final ByteData? byteData = await image.toByteData(format:
ImageByteFormat.png);
  if (byteData != null) {
    final Uint8List pngBytes = byteData.buffer.asUint8List();
    // Get the application's document directory
    final directory = await getApplicationDocumentsDirectory();
    final filePath = '${directory.path}/qr_code.png';
    // Write the image data to the file
    final file = File(filePath);
    await file.writeAsBytes(pngBytes);
    print("QR Code saved at: $filePath");
  } else {
    print("Failed to convert image to ByteData.");
  }
}
```

Figure 4.27 Saving QR code to file

The file of the QR code is later retrieved and is displayed in the profile page as an image file. The code for retrieving the QR code is shown in the figure below.

```dart
//Retrieve the user's qr code from the file
Future<void> RetrieveQrCode () async {
  final directory = await getApplicationDocumentsDirectory();
  print(directory);
  final filePath = '${directory.path}/qr_code.png';
  setState(() {
    QrCodeImage = File(filePath);
  });
}
```

Figure 4.28 Retrieving the QR code

The user can scan for QR codes alongside having their own. This technique is implemented using the mobile_scanner package. The code in the figure below shows how the mobile_scanner works.

```dart
MobileScanner (
            controller: camController,
            onDetect: (capture) async {
            if(QrScanner.isScanned){
              final List<Barcode> barcodes = capture.barcodes;
              for (final barcode in barcodes) {
                 debugPrint('Barcode found! ${barcode.rawValue}');
                 //add new contact
                 await AddContact(barcode.rawValue.toString());
                 if(pop){
                   Navigator.pop(context,true);
                 }
              }
           }
         }
       ),
QRScannerOverlay(
            overlayColor: Colors.white12.withOpacity(0.5)
)
```

Figure 4.29 QR code scanner

C. Shared preferences

The main objective of this project is to configure a system that can send and receive data in remote areas, and thus, the application should not be dependent on any external network (e.g. Wi-Fi). Consequently, we cannot use external databases that are accessible wirelessly through other networks like Wi-Fi. Therefore, we have used a flutter dependency called shared preferences to create a storage space on the device's local storage. Thus, the data is stored and fetched locally, and our application is independent of any external connection. All the data related to a user's account is stored in shared preferences. However, it is important to note that once the application is uninstalled on the device, all the data will be deleted from the device's local storage space configured by shared preferences.

First, an instance of shared preferences must be initialized. Following that, every data stored is fetched using the shared preferences object. The initialization is displayed in the figure below.

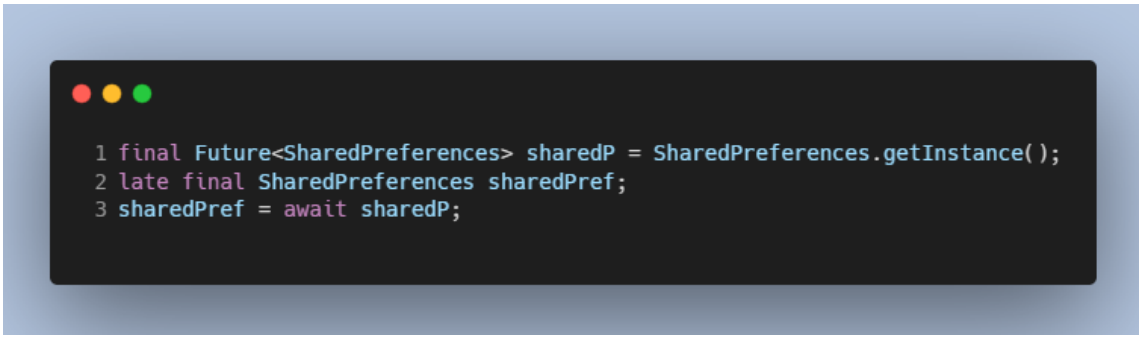


Figure 4.30 Shared preferences

In this project, shared preferences store three main elements:

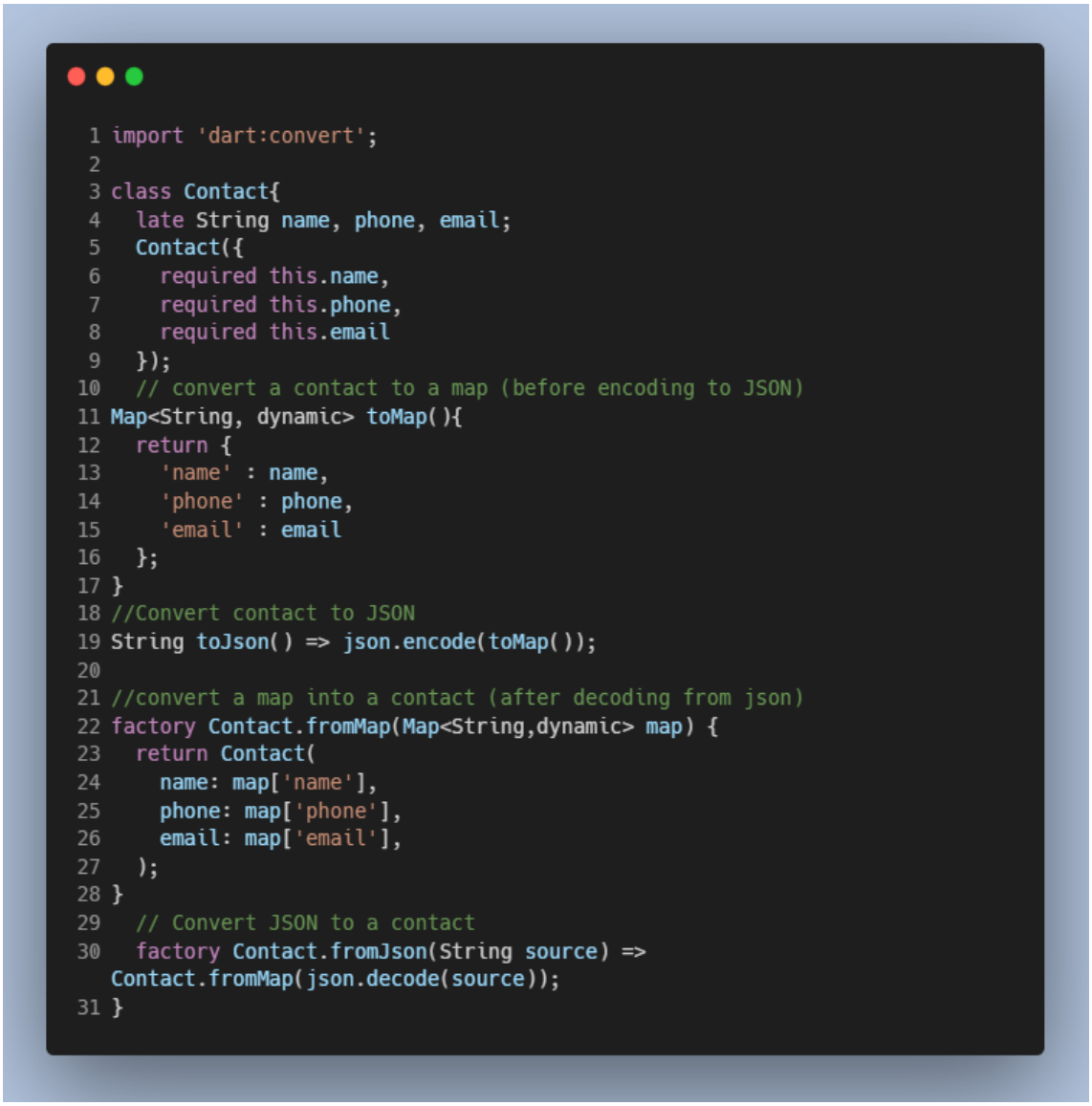
- *isSignedIn*: this is a user-defined Boolean value that is assigned to *true* once the user account is created. Before the user creates an account, *isSignedIn* is false and, the user is navigated to the sign-up page. Once the user account is created, the variable's value is changed to true, and this way, the user will never be navigated to the sign-up page again.

- User account: when the user's account is created, all data related to the us-

er is stored in a separate entity in shared preferences called "*userAdded*"

- Data entities: the types of data in this application are modeled into three different classes: Contact, Message, and Chat. All instances of these classes are stored in shared preferences.

Instances of these classes are constantly added, edited, or removed from our database. All the logic behind these methods is explained later in this part. Meanwhile, the codes for the three classes are shown in the figures below.



```dart
import 'dart:convert';

class Contact{
  late String name, phone, email;
  Contact({
    required this.name,
    required this.phone,
    required this.email
  });
  // convert a contact to a map (before encoding to JSON)
Map<String, dynamic> toMap(){
  return {
    'name' : name,
    'phone' : phone,
    'email' : email
  };
}
//Convert contact to JSON
String toJson() => json.encode(toMap());

//convert a map into a contact (after decoding from json)
factory Contact.fromMap(Map<String,dynamic> map) {
  return Contact(
    name: map['name'],
    phone: map['phone'],
    email: map['email'],
  );
}
  // Convert JSON to a contact
  factory Contact.fromJson(String source) =>
Contact.fromMap(json.decode(source));
}
```

Figure 4.31 Contact class

This is the class for contacts that are added to our system. The class contains the

main parameters stored in the QR code of our contacts when they are scanned.

The method toMap() is used to convert an instance of the Contact class to a map instance where each value of the class object is represented by a key-value pair in the map instance.

The method fromMap() does the opposite where it converts each map instance to an object instance for the Contact class, and each key value is mapped to its corresponding value in the class object.

```dart
class Message{
  String? senderID;
  String? content;
  DateTime? sentAt;

  Message({
    required this.senderID,
    required this.content,
    required this.sentAt,
  });

  Message.fromJson(Map<String, dynamic> json) {
    senderID = json['senderID'];
    content = json['content'];
    sentAt = DateTime.parse(json['sentAt']);
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = <String, dynamic>{};
    data['senderID'] = senderID;
    data['content'] = content;
    data['sentAt'] = sentAt!.toIso8601String();
    return data;
  }
}
```

Figure 4.32 Message Class

This class represents a message instance. For each message created between users, the sender's identifier, the message content, and the time it was sent are documented and saved as an instance to shared preferences. The methods, toJson(), and fromJson() do the same function as the toMap() and fromMap() methods in the Contact class.

```dart
import 'package:lora_chat/Contacts.dart';
import 'package:lora_chat/Messages.dart';

class Chat {
  String? id;
  List<Contact>? participants;
  List<Message>? messages;

  Chat({
    required this.id,
    required this.participants,
    required this.messages,
  });

  Chat.fromJson(Map<String, dynamic> json) {
    id = json['id'];
    participants = List.from(json['participants']).map((m) =>
Contact.fromJson(m)).toList();
    messages =
        List.from(json['messages']).map((m) =>
Message.fromJson(m)).toList();
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = <String, dynamic>{};
    data['id'] = id;
    data['participants'] = participants?.map((m) => m.toJson()).toList()
?? [];
    data['messages'] = messages?.map((m) => m.toJson()).toList() ?? [];
    return data;
  }
}
```

Figure 4.33 Chat class

This class represents a chat instance. For each contact added, a chat instance is created between the contact and our user. Each chat instance must contain a chat identifier (the calculation behind creating the chat identifier is explained later in this part), Contact instances of both parties in the chat, and a list of all the messages shared between both users. Similar to the Message and Contact class, the Chat class has toJson(), and fromJson() methods which convert a Chat instance to a map instance and vice versa.

These class models are the main data stored in our application. Instances of these classes are continuously created, modified, deleted, and listed throughout the application's operation. The codes explained in the figures below depict the function-

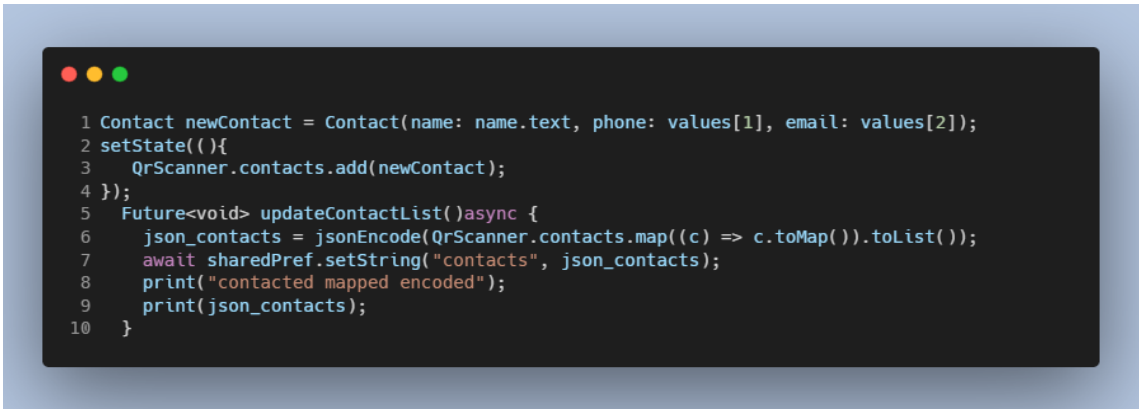
alities of these data models and their usage.

When the user account is initially created, the information regarding the user is stored in the *“userAdded”* instance of *sharedPref.* This instance is created once and never modified. The account data are later displayed in the *ProfilePage* in the application. The method displayed in the figure below is called in the *SignUp* page once to store the user’s data in *“userAdded”*.

Figure 4.34 Add account data to sharedPref

Subsequently, when the user is created, two empty lists are created, one for the contacts, and the other for the chats. A contact can be added by scanning their QR codes. Once a QR code is scanned, a Contact instance is created for the scanned user and is added to the contact list. Correspondingly a Chat instance is created for the scanned user and our user. An added contact instance can be deleted if the delete button is pressed, and once deleted, automatically, the Chat instance related to them is deleted as well.

The figure below shows the method responsible for adding the contact list to *sharedPref*. Notice that the map function iterates through all Contact instances in the *contacts* list and converts them to Map instances using the toMap() method, and the result is converted to a list, and then this value is encoded to a JSON string using the *JsonEncode* method. Finally, the JSON string created is stored in *sharedPref* with the key *“contacts”*.



```
Contact newContact = Contact(name: name.text, phone: values[1], email: values[2]);
setState((){
   QrScanner.contacts.add(newContact);
});
  Future<void> updateContactList()async {
    json_contacts = jsonEncode(QrScanner.contacts.map((c) => c.toMap()).toList());
    await sharedPref.setString("contacts", json_contacts);
    print("contacted mapped encoded");
    print(json_contacts);
  }
```

Figure 4.35 Add contact list to *sharedPref*

Similarly, a Chat instance is created, the chat list is updated, and stored to *sharedPref*. The figures below show how a Chat instance is created, and how the chat identifier is generated.

```
  Future<void> CreateChat(Contact newContact) async {
    print("create chat");
    String chatID = CreateID(newContact);
    Chat newChat = Chat(id: chatID,
        participants: [newContact, Contact(name:
jsonDecode(sharedPref.getString("userAdded").toString())[1],
            phone: jsonDecode(sharedPref.getString("userAdded").toString())[0],
            email: jsonDecode(sharedPref.getString("userAdded").toString())[2])
          ],
        messages: []);
    setState(() {
        QrScanner.chats.add(newChat);
    });
  }
```

Figure 4.36 create a chat instance method

```
  String CreateID(Contact newContact){
    List phones = [newContact.phone,
jsonDecode(sharedPref.getString("userAdded").toString())[0]];
    phones.sort();
    String chatID = phones.fold("", (init,acc) => "$init$acc");
    print("chatID");
    print(chatID);
    return chatID;
  }
```

Figure 4.37 create a chat identifier method

The *CreateID* method takes the identifiers of both users, sorts them in ascending order, and concatenates them to create the chat identifier. The retrieved value of the *CreateID* method is assigned to the *chatID* variable in the *CreateChat* method. Then, a new Chat instance is created with the returned *chatID* value as the identifier, and the new user with our user as participants. On Chat creation, the *messages* list is kept empty. After the Chat instance is created, it's added to the *chats* list, and then the list is stored to *sharedPref* in the method depicted in the figure below. The *chats* list is stored in a manner identical to the *contacts* list.

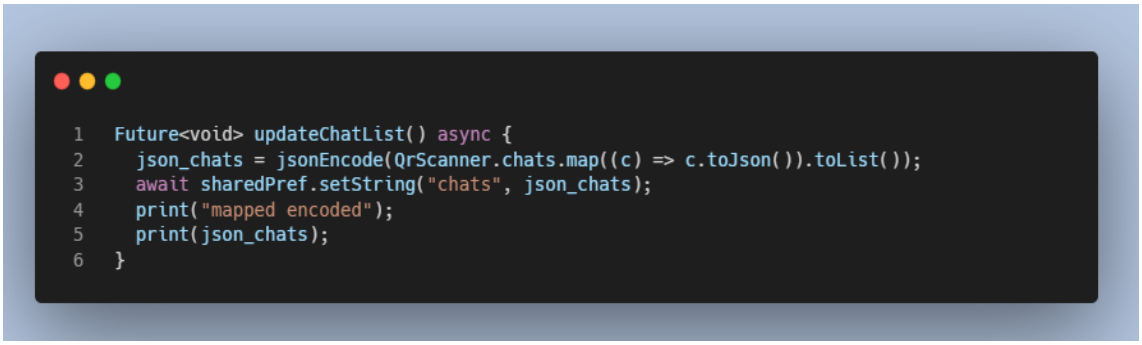


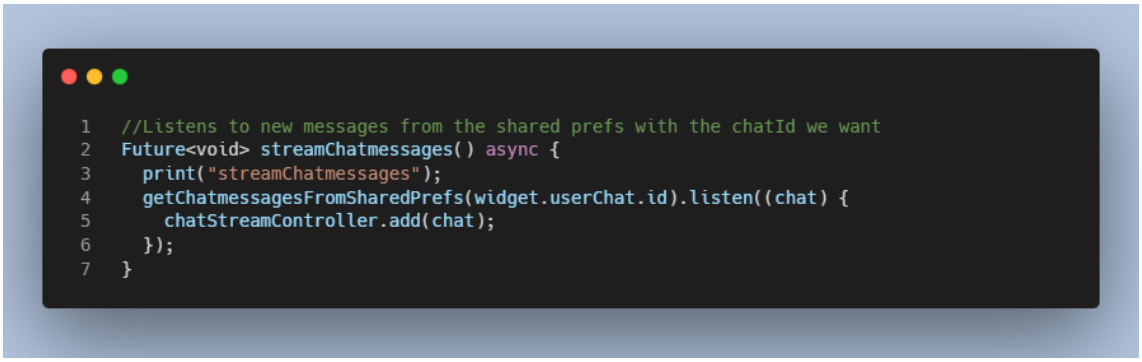
Figure 4.38 add chat list to *sharedPref*

As soon as the Chat instance is created, the users can enter the chat screen and start chatting. Several methods are used to store, stream, update, and retrieve chat messages. Initially, when the user is navigated to the chat screen, the initState() method is called. *initState* method belongs to the stateful widget's object *"State"*. Furthermore, the initState() method is called once at the creation of the *"State"* object. Thus, this method is useful for initializing variables, and setting up streams. Subsequently, in the initState() method of our *"State"* object, two instances of *ChatUser* are created, one for each participant in the chat. Besides that, the streamChatMessages() method is called to listen to the *chatStreamController* stream. The figure below shows the initState() method.

```
@override
void initState() {
  super.initState();
  // id = phone number
  me = ChatUser(id: jsonDecode(sharedPref.getString("userAdded").toString())[0],
      firstName: jsonDecode(sharedPref.getString("userAdded").toString())[1]);
  otherUser = ChatUser(id: widget.userChat.participants![0].phone,
      firstName: widget.userChat.participants![0].name);
  // start streaming the messages
  initialize();
}
Future<void> initialize() async {
  await streamChatmessages();
  await readData();
}
```

Figure 4.39 initState() method

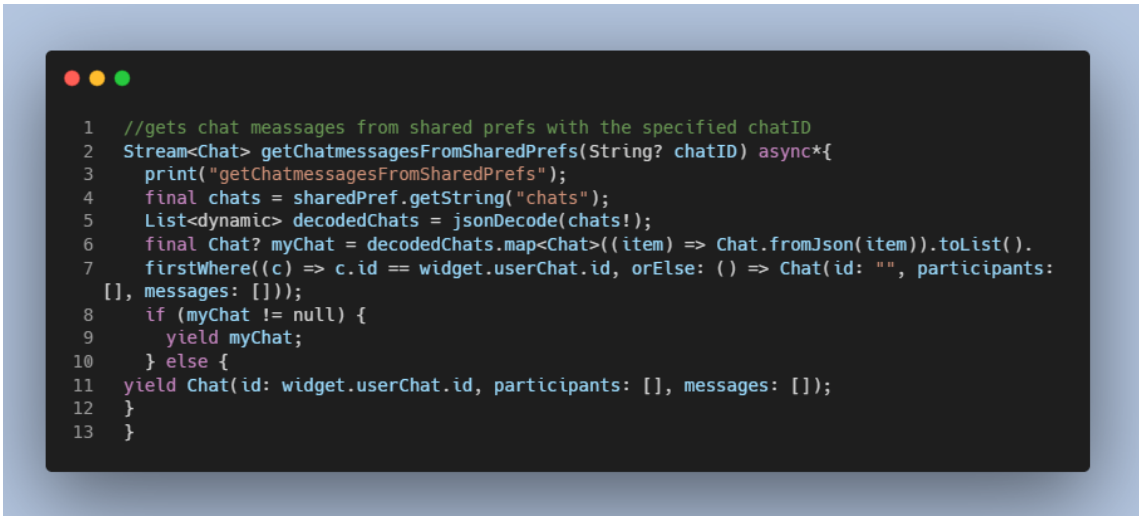
The streamChatMessages() method listens to a user-defined Chat stream created in the method *getChatmessagesFromSharedPrefs* which has a return type of *Stream<Chat>*, and retrieves chat messages of the chat based on its identifier from *sharedPref*. Therefore, the streamChatMessages() method listens for new messages in the chat and adds them to the *chatStreamController* stream. *chatStreamController is* a stream controller created in our *"State"* object class. The figures below display both the *getChatmessagesFromSharedPrefs* and stream-ChatMessages() methods.



```
//Listens to new messages from the shared prefs with the chatId we want
Future<void> streamChatmessages() async {
  print("streamChatmessages");
  getChatmessagesFromSharedPrefs(widget.userChat.id).listen((chat) {
    chatStreamController.add(chat);
  });
}
```

Figure 4.40 streamChatMessages() method

This method calls the *getChatmessagesFromSharedPrefs* method with the current chat's *chatID* to listen for new messages added to the *sharedPref* for the intended chat.



```dart
//gets chat meassages from shared prefs with the specified chatID
Stream<Chat> getChatmessagesFromSharedPrefs(String? chatID) async*{
  print("getChatmessagesFromSharedPrefs");
  final chats = sharedPref.getString("chats");
  List<dynamic> decodedChats = jsonDecode(chats!);
  final Chat? myChat = decodedChats.map<Chat>((item) => Chat.fromJson(item)).toList().
  firstWhere((c) => c.id == widget.userChat.id, orElse: () => Chat(id: "", participants:
 [], messages: []));
  if (myChat != null) {
    yield myChat;
  } else {
 yield Chat(id: widget.userChat.id, participants: [], messages: []);
 }
 }
```

Figure 4.41 getChatmessagesFromSharedPrefs method

This method retrieves the chats list from *sharedPref,* and searches for the chat with the intended chatID. If found, it returns the chat, and if not, it returns an empty chat. Recall that when we added the chats list to *sharedPref*, we converted all Chat instances to Map instances and encoded the new list of Map instances to a JSON string, and then we stored it in our database. Notice that when we retrieve the chats from *sharedPref* in the method above, the JSON string is decoded to a List of Map instances. After that, the *map* method iterates through all the instances and converts them to Chat instances using the fromJson() method.

Chat messages are saved to chat instances in *sharedPref* using the *saveChatMessage* method. This method is called when a message is sent to or received from a user of the chat.

```dart
// save messages to shared preferences
Future<void> saveChatMessage(Message message)async {
  widget.userChat.messages!.add(message);
  json_chats = jsonEncode(QrScanner.chats.map((c) => c.toJson()).toList());
  await sharedPref.setString("chats", json_chats);
  streamChatmessages();
}
```

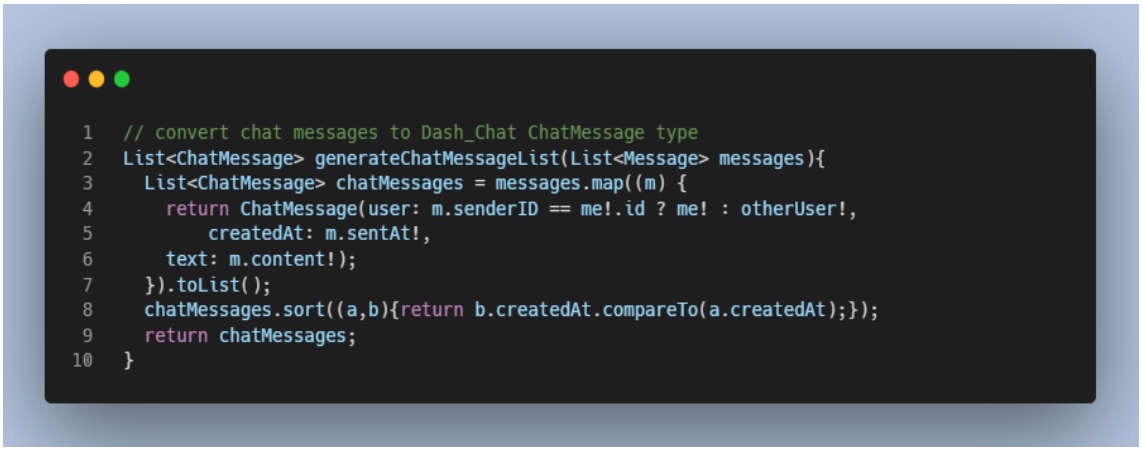
Figure 4.42 *saveChatMessage* method

After ensuring that all messages are successfully saved and stored on the database, we need methods that retrieve these messages and print them on the screen. The method retrieveChatmessages() depicted in the figure below returns the data stream from the *chatStreamController* stream.

```
// for displaying
Stream<Chat> retrieveChatmessages(){
  print("retrieveChatmessages");
  return chatStreamController.stream;
}
```

Figure 4.43 retrieveChatmessages() method

This method is assigned to the *stream* parameter in the *streamBuilder* of our *"State"* object class. Therefore, *streamBuilder* listens to all the new messages added to the *chatStreamController.* However, since the chat box is created by the dash_chat_2, the messages of type Message in the *chatStreamController* need to be converted to *ChatMessage* type of the dash_chat_2. This process is done by the *generateChatMessageList* method shown in the figure below.



```
// convert chat messages to Dash_Chat ChatMessage type
List<ChatMessage> generateChatMessageList(List<Message> messages){
  List<ChatMessage> chatMessages = messages.map((m) {
    return ChatMessage(user: m.senderID == me!.id ? me! : otherUser!,
        createdAt: m.sentAt!,
    text: m.content!);
  }).toList();
  chatMessages.sort((a,b){return b.createdAt.compareTo(a.createdAt);});
  return chatMessages;
}
```

Figure 4.44 generateChatMessageList method

The returned value from this method is printed on the chat screen. In addition, this method also sorts messages based on the time they were sent, so they're printed on the screen in the right order.

D. Alert dialogs

Alert dialogs are used throughout the application to warn users of malfunctions that might suddenly occur. Particularly the fact that this application is built on BLE connection and message processing, the absence of connection makes the application completely impractical. Therefore, we have provided the user with alert dialogs to warn them in case of a breakdown in one of the services provided by the application. Besides that, alert dialogs are also used to display information or ask the user to take some action. The alert dialogs used in our application are listed below with their codes:

- If the user opens the application without turning on the Bluetooth button in the mobile phone, an alert is displayed on the screen. The figure below shows the code for this alert.

```
if (state == !BluetoothState.on) {
          return Center(
            child: ShowDialog(context)
            );
        }

  ShowDialog(BuildContext context) {
    return AlertDialog(
        //backgroundColor: Colors.white,
        shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
        backgroundColor: Colors.grey[200],
        contentPadding: const EdgeInsets.all(20),
        title: const Text("Warning!"),
        content: const Text("Please turn the bluetooth on",
        style: TextStyle(
            color: Colors.black)
    ),
    );
  }
```

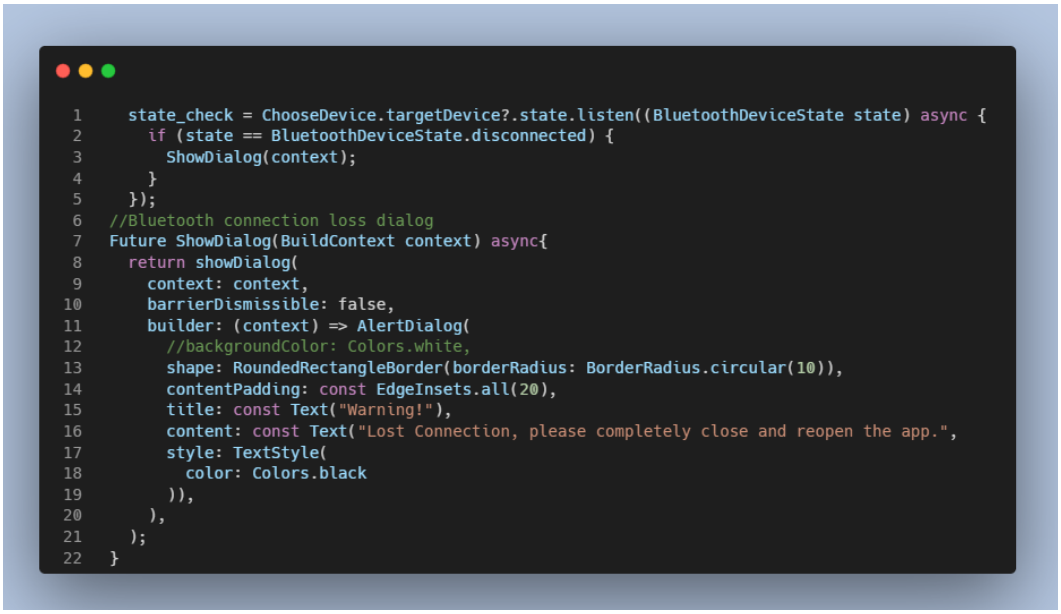
Figure 4.45 Alert dialog

- When the user creates an account and presses the sign-up button, if the user fills all the fields correctly, the user account is created and a dialog is displayed showing that the user is created successfully. The figure below shows this dialog.

```
onPressed: () async {
               if(formKey.currentState!.validate()){
                  ShowDialog(context);
                } else{
                    // form is not submitted till all cells are filled
                }
             }
Future ShowDialog(BuildContext context) async{
    return showDialog(
        context: context,
        barrierDismissible: false,
        builder: (context) => AlertDialog(
            backgroundColor: Colors.white.withOpacity(0.8),
            shape: RoundedRectangleBorder(borderRadius: BorderRadius.all(Radius.circular(20))),
            content:  const Text("User created",
                style: TextStyle(
                    fontSize: 16,
                    color: Colors.black87 ))
        )
    );
  }
```

Figure 4.46 Alert dialog

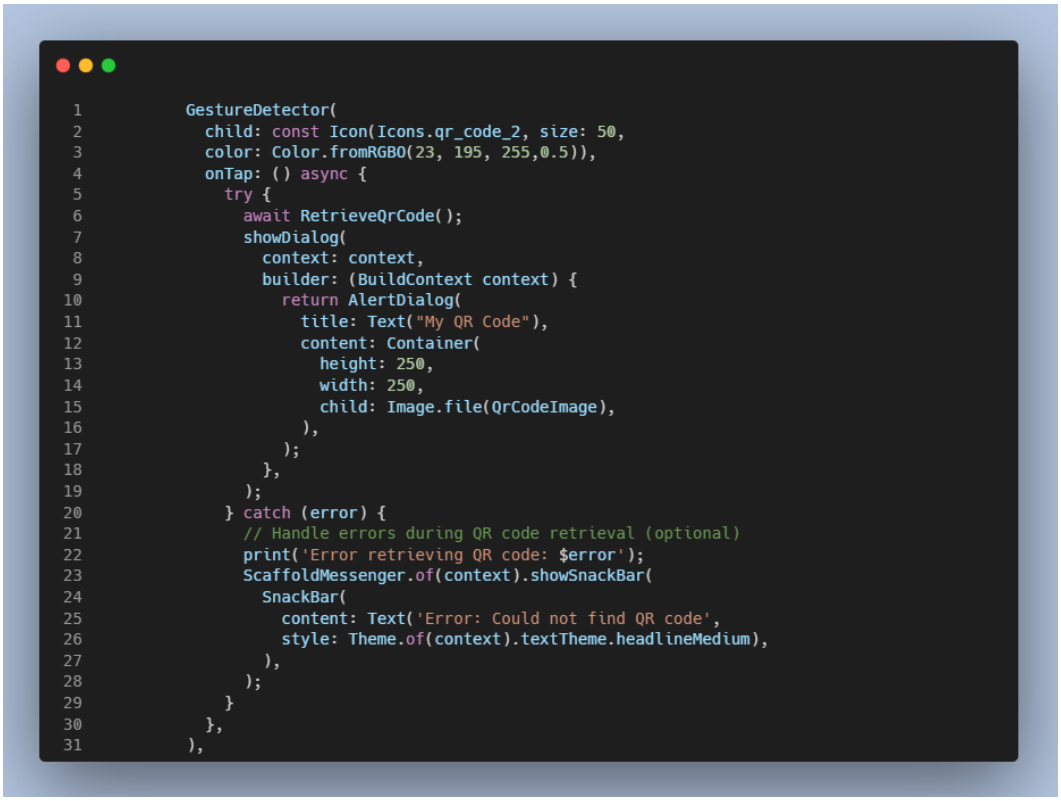
- If a Bluetooth LE connection is lost anytime during the application runtime, a dialog is displayed to warn the user to check why the connection is lost. As well as that, to reconnect to the same Bluetooth device, the user must close the application completely and then open it again. The figure below depicts the dialog.



```
    state_check = ChooseDevice.targetDevice?.state.listen((BluetoothDeviceState state) async {
      if (state == BluetoothDeviceState.disconnected) {
        ShowDialog(context);
      }
    });
  //Bluetooth connection loss dialog
  Future ShowDialog(BuildContext context) async{
    return showDialog(
      context: context,
      barrierDismissible: false,
      builder: (context) => AlertDialog(
        //backgroundColor: Colors.white,
        shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
        contentPadding: const EdgeInsets.all(20),
        title: const Text("Warning!"),
        content: const Text("Lost Connection, please completely close and reopen the app.",
        style: TextStyle(
          color: Colors.black
        )),
      ),
    );
  }
```

Figure 4.47 Alert dialog

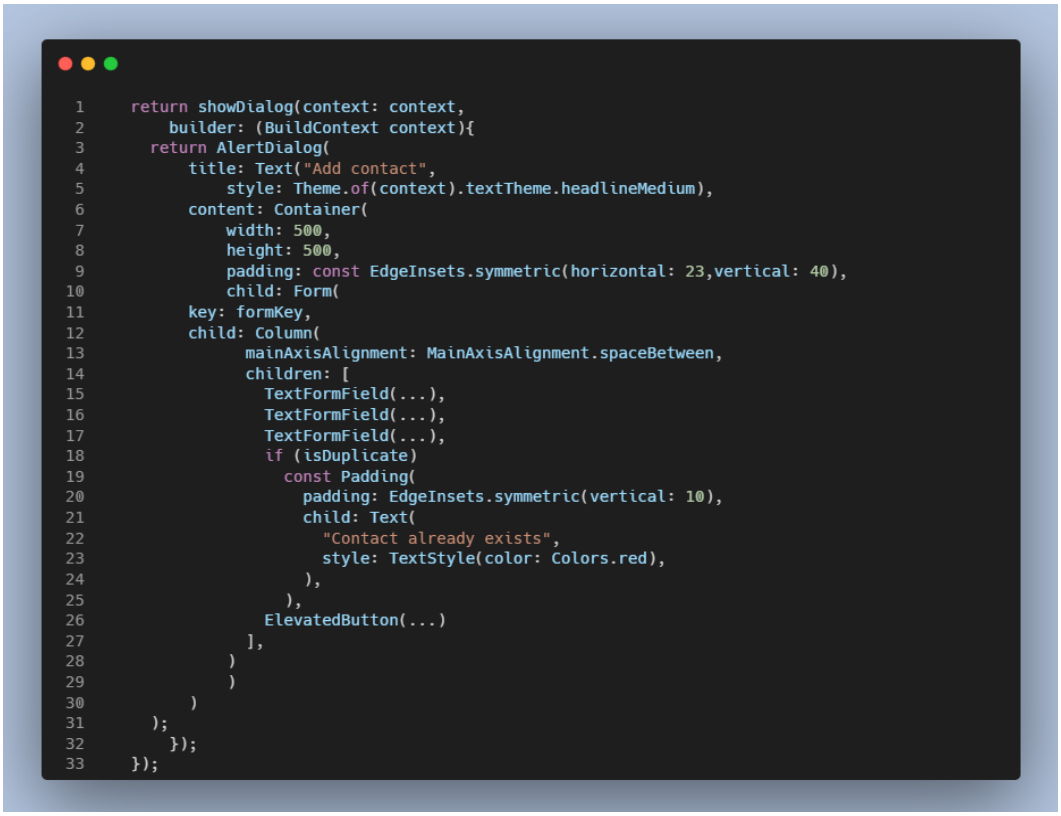
- In the profile page, when tapping on the QR code icon, a dialog is opened that displays the QR code of the user. This dialog is used when a user wants to add another user to the contacts and therefore needs to scan their QR code. The figure below shows this dialog.



```
GestureDetector(
  child: const Icon(Icons.qr_code_2, size: 50,
  color: Color.fromRGBO(23, 195, 255,0.5)),
  onTap: () async {
    try {
      await RetrieveQrCode();
      showDialog(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: Text("My QR Code"),
            content: Container(
              height: 250,
              width: 250,
              child: Image.file(QrCodeImage),
            ),
          );
        },
      );
    } catch (error) {
      // Handle errors during QR code retrieval (optional)
      print('Error retrieving QR code: $error');
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('Error: Could not find QR code',
          style: Theme.of(context).textTheme.headlineMedium),
        ),
      );
    }
  },
),
```

Figure 4.48 Alert dialog

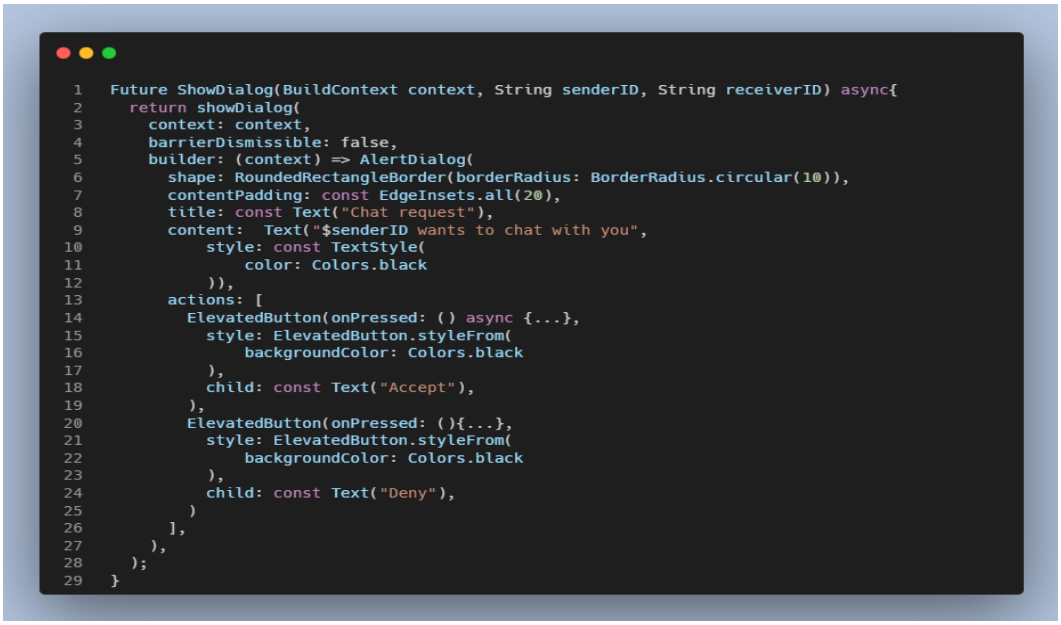
- When a user's QR code is scanned, an alert dialog is popped that displays the data retrieved from the QR code which are the name, phone number, and email address in three different text fields. Only the name of the user can be edited, but the email address and phone number are fixed values. The user presses the *save* button to save the contact and the dialog is popped. It's important to note that the Boolean value *isDuplicate* is used to prevent the user from saving a contact that already exists in their contact list by disabling the *save* button.



```
return showDialog(context: context,
    builder: (BuildContext context){
  return AlertDialog(
      title: Text("Add contact",
          style: Theme.of(context).textTheme.headlineMedium),
      content: Container(
          width: 500,
          height: 500,
          padding: const EdgeInsets.symmetric(horizontal: 23,vertical: 40),
          child: Form(
      key: formKey,
      child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: [
              TextFormField(...),
              TextFormField(...),
              TextFormField(...),
              if (isDuplicate)
                const Padding(
                  padding: EdgeInsets.symmetric(vertical: 10),
                  child: Text(
                    "Contact already exists",
                    style: TextStyle(color: Colors.red),
                  ),
                ),
              ElevatedButton(...)
            ],
          )
          )
      )
  );
    });
});
```

Figure 4.49 Alert dialog

- When a key request data packet is sent to the user, a dialog is displayed to ask the user whether they want to accept or deny the request. The dialog for this case is displayed in the figure below.



```
Future ShowDialog(BuildContext context, String senderID, String receiverID) async{
  return showDialog(
    context: context,
    barrierDismissible: false,
    builder: (context) => AlertDialog(
      shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
      contentPadding: const EdgeInsets.all(20),
      title: const Text("Chat request"),
      content:  Text("$senderID wants to chat with you",
          style: const TextStyle(
              color: Colors.black
          )),
      actions: [
        ElevatedButton(onPressed: () async {...},
          style: ElevatedButton.styleFrom(
              backgroundColor: Colors.black
          ),
          child: const Text("Accept"),
        ),
        ElevatedButton(onPressed: (){...},
          style: ElevatedButton.styleFrom(
              backgroundColor: Colors.black
          ),
          child: const Text("Deny"),
        )
      ],
    ),
  );
}
```

Figure 4.50 Alert dialog

4.3.2 Frontend Development

This section Iterates through the frontend of our application, discusses the codes, and relates them to the backend. The frontend's main functionality is to give the user access to the backend functionalities. However, a utilized frontend for an application is also crucial to give the user a better understanding and more comfortable experience with the application.

In Flutter, all the components displayed on a screen are wrapped in widgets. Widgets include buttons, text fields, containers, …etc. These widgets must belong to an upper widget class that is either a stateless widget or a stateful widget. A stateless widget has an immutable state meaning that the state of the widget does not change during the widget lifecycle. A stateful widget, however, has a mutable state indicating that the state may change at a point during the widget lifecycle. Whether stateless or stateful, a state widget must return the method build. The *build* method takes a *BuildContext* as an argument and returns a *Widget* tree that displays the user interface. While the build method is called only once in the stateless widget, it can be called multiple times in a stateful widget.

To develop a uniform and systematic interface, we have initialized a colour scheme, text themes, and button styles that are predefined for the entire application. This ensures that all the pages are well structured. The figure below provides the code for the application style format.

```
        theme: ThemeData(
          scaffoldBackgroundColor: const Color.fromRGBO(41, 41, 41,0.5),
          useMaterial3: true,
          colorSchemeSeed: const Color.fromRGBO(41,41,41,0.5),
          elevatedButtonTheme: ElevatedButtonThemeData(
            style: ButtonStyle(
                padding: WidgetStateProperty.all(const EdgeInsets.symmetric(horizontal: 20, vertical: 10)),
                textStyle: WidgetStateProperty.all(TextStyle(fontSize: 16)),
                shape: WidgetStateProperty.all(RoundedRectangleBorder(
                  borderRadius: BorderRadius.circular(20),
            )
          )
        )
      ),
          textTheme: const TextTheme(
            headlineLarge: TextStyle(fontSize: 32, fontWeight: FontWeight.bold, color: Color.fromRGBO(23, 195, 255,0.5)),
            headlineMedium: TextStyle(fontSize: 22, color: Color.fromRGBO(23, 195, 255,0.5)),
            bodyMedium: TextStyle(fontSize: 16, color: Color.fromRGBO(23, 195, 255,0.5)),
            bodySmall: TextStyle(fontSize: 12, color: Color.fromRGBO(23, 195, 255,0.5)),
            bodyLarge: TextStyle(fontSize: 20, color: Color.fromRGBO(23, 195, 255,0.5))
          ),
          appBarTheme: const AppBarTheme(
            backgroundColor: Colors.transparent,
            elevation: 0,
            titleTextStyle: TextStyle(fontSize: 36, color: Color.fromRGBO(158, 195, 253,0.5)),
          )
    )
```

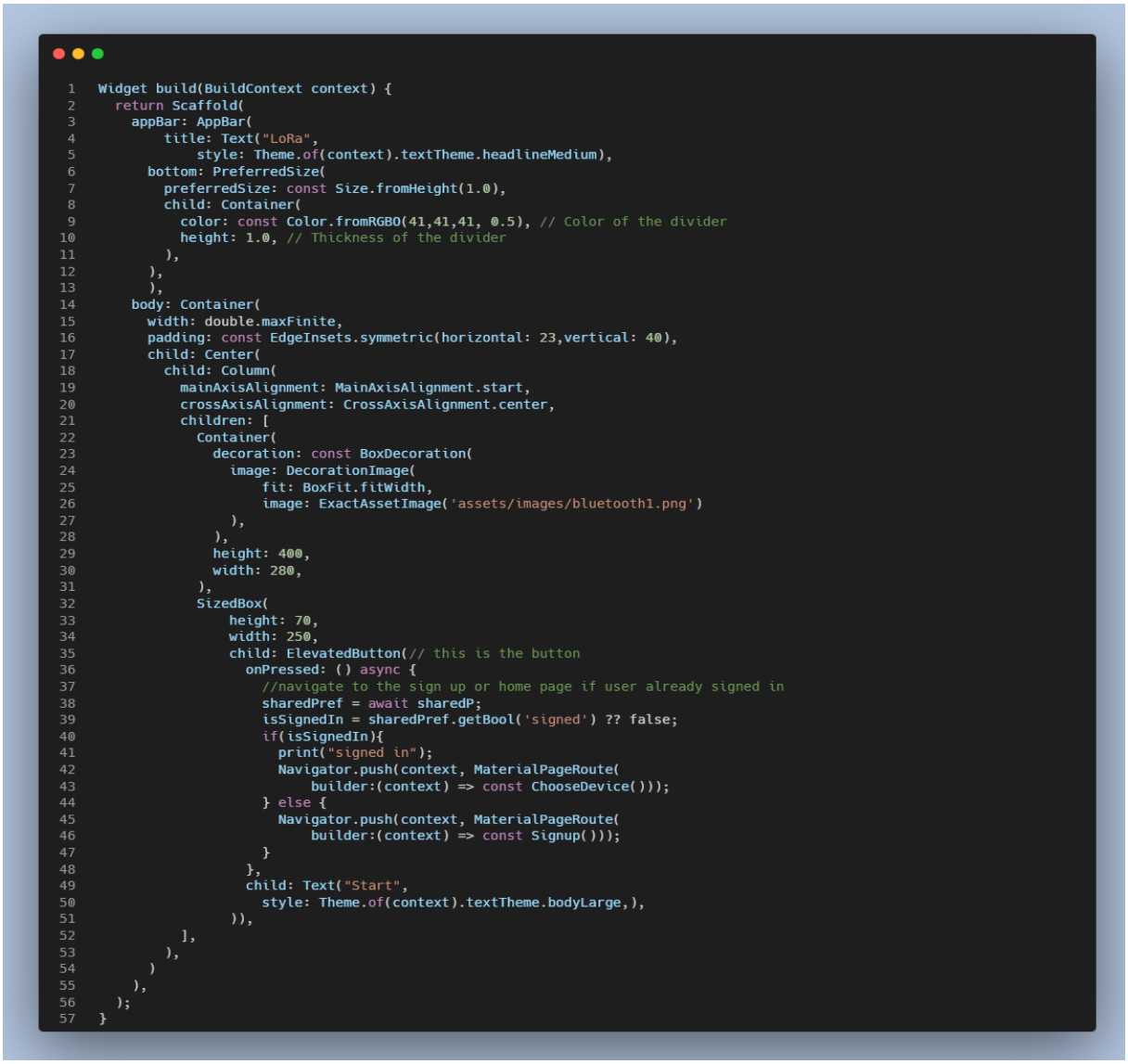
Figure 5.51 application theme style

The starting page of the application is a screen with a start button to navigate to the application. However, if the user has not turned on the Bluetooth scan button on the mobile phone, an alert dialog is displayed, and when they turn the Bluetooth on, the dialog is popped, and the starting page is displayed again. Furthermore, If the user is already signed up, they are navigated to the Bluetooth connect page, if the user is not signed up yet, they are navigated to the sign-up page. The figure below shows the code for the starting page.

```
Widget build(BuildContext context) {
  return StreamBuilder<BluetoothState>(
    stream: FlutterBlue.instance.state,
    initialData: BluetoothState.unknown,
    builder: (c, snapshot) {
      final state = snapshot.data;
      if (state == BluetoothState.on) {
        return const LandingPage();
      } else {
        return Center(
          child: ShowDialog(context)
        );
      }
    });
}
```

Figure 5.52 starting page build method

The build method for the starting page returns the landing page if the condition is met, and if not, it returns the alert dialog. The figure below shows the landing page's build method.



```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
        title: Text("LoRa",
            style: Theme.of(context).textTheme.headlineMedium),
      bottom: PreferredSize(
        preferredSize: const Size.fromHeight(1.0),
        child: Container(
          color: const Color.fromRGBO(41,41,41, 0.5), // Color of the divider
          height: 1.0, // Thickness of the divider
        ),
      ),
      ),
    body: Container(
      width: double.maxFinite,
      padding: const EdgeInsets.symmetric(horizontal: 23,vertical: 40),
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.start,
          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
            Container(
              decoration: const BoxDecoration(
                image: DecorationImage(
                    fit: BoxFit.fitWidth,
                    image: ExactAssetImage('assets/images/bluetooth1.png')
                ),
              ),
              height: 400,
              width: 280,
            ),
            SizedBox(
                height: 70,
                width: 250,
                child: ElevatedButton(// this is the button
                  onPressed: () async {
                    //navigate to the sign up or home page if user already signed in
                    sharedPref = await sharedP;
                    isSignedIn = sharedPref.getBool('signed') ?? false;
                    if(isSignedIn){
                      print("signed in");
                      Navigator.push(context, MaterialPageRoute(
                          builder:(context) => const ChooseDevice()));
                    } else {
                      Navigator.push(context, MaterialPageRoute(
                          builder:(context) => const Signup()));
                    }
                  },
                  child: Text("Start",
                    style: Theme.of(context).textTheme.bodyLarge,),
                )),
          ],
        ),
      )
    ),
  );
}
```

Figure 5.53 landing page build method

The figure below shows the UI for the landing page.

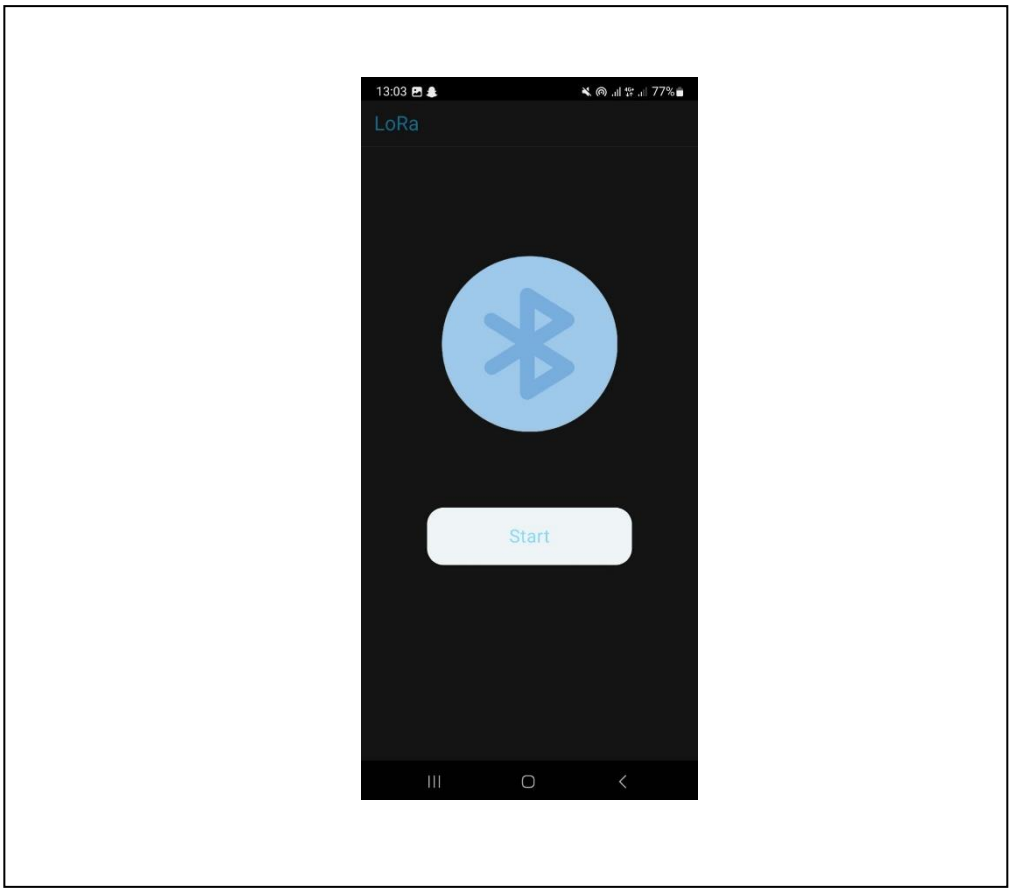
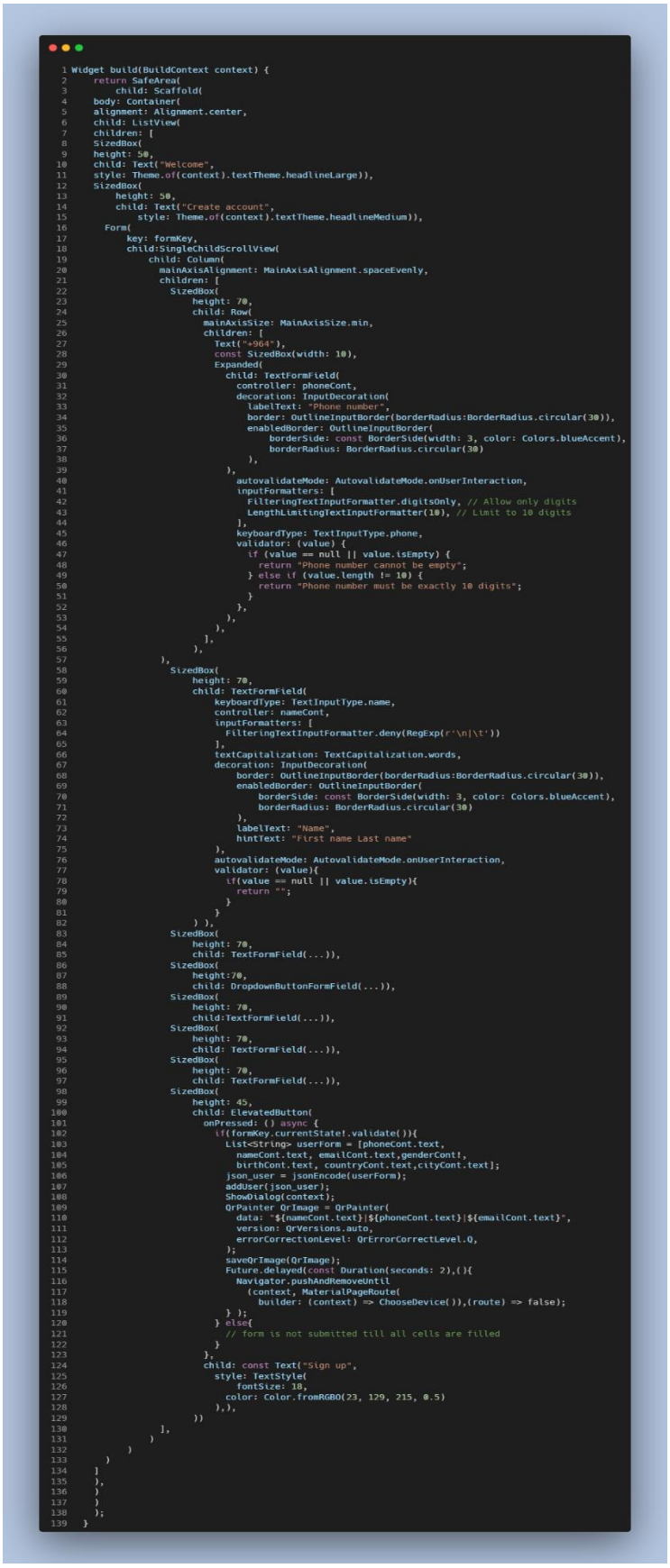


Figure 4.54 landing page UI

Supposing that the user has not signed up yet, they will be navigated to the sign-up page. The build method for the sign-up page is displayed in the figure below. In the figure below only two text fields are displayed one that implements a drop-down form field, and the other is a normal text field. The other text fields' content is hidden since they are all similar to the shown text field.



```
Widget build(BuildContext context) {
    return SafeArea(
        child: Scaffold(
    body: Container(
    alignment: Alignment.center,
    child: ListView(
    children: [
    SizedBox(
    height: 50,
    child: Text("Welcome",
    style: Theme.of(context).textTheme.headlineLarge)),
    SizedBox(
        height: 50,
        child: Text("Create account",
          style: Theme.of(context).textTheme.headlineMedium)),
      Form(
          key: formKey,
          child:SingleChildScrollView(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.spaceEvenly,
                children: [
                  SizedBox(
                      height: 70,
                      child: Row(
                        mainAxisSize: MainAxisSize.min,
                        children: [
                          Text("+964"),
                          const SizedBox(width: 10),
                          Expanded(
                            child: TextFormField(
                              controller: phoneCont,
                              decoration: InputDecoration(
                                labelText: "Phone number",
                                border: OutlineInputBorder(borderRadius:BorderRadius.circular(30)),
                                enabledBorder: OutlineInputBorder(
                                    borderSide: const BorderSide(width: 3, color: Colors.blueAccent),
                                    borderRadius: BorderRadius.circular(30)
                                ),
                              ),
                              autovalidateMode: AutovalidateMode.onUserInteraction,
                              inputFormatters: [
                                FilteringTextInputFormatter.digitsOnly, // Allow only digits
                                LengthLimitingTextInputFormatter(10), // Limit to 10 digits
                              ],
                              keyboardType: TextInputType.phone,
                              validator: (value) {
                                if (value == null || value.isEmpty) {
                                  return "Phone number cannot be empty";
                                } else if (value.length != 10) {
                                  return "Phone number must be exactly 10 digits";
                                }
                              },
                            ),
                          ),
                        ],
                      ),
                  ),
                    SizedBox(
                        height: 70,
                        child: TextFormField(
                            keyboardType: TextInputType.name,
                            controller: nameCont,
                            inputFormatters: [
                              FilteringTextInputFormatter.deny(RegExp(r'\n|\t'))
                            ],
                            textCapitalization: TextCapitalization.words,
                            decoration: InputDecoration(
                                border: OutlineInputBorder(borderRadius:BorderRadius.circular(30)),
                                enabledBorder: OutlineInputBorder(
                                    borderSide: const BorderSide(width: 3, color: Colors.blueAccent),
                                    borderRadius: BorderRadius.circular(30)
                                ),
                                labelText: "Name",
                                hintText: "First name Last name"
                            ),
                            autovalidateMode: AutovalidateMode.onUserInteraction,
                            validator: (value){
                              if(value == null || value.isEmpty){
                                return "";
                              }
                            }
                        ) ),
                    SizedBox(
                        height: 70,
                        child: TextFormField(...)),
                    SizedBox(
                        height:70,
                        child: DropdownButtonFormField(...)),
                    SizedBox(
                        height: 70,
                        child:TextFormField(...)),
                    SizedBox(
                        height: 70,
                        child: TextFormField(...)),
                    SizedBox(
                        height: 70,
                        child: TextFormField(...)),
                    SizedBox(
                        height: 45,
                        child: ElevatedButton(
                          onPressed: () async {
                            if(formKey.currentState!.validate()){
                              List<String> userForm = [phoneCont.text,
                                nameCont.text, emailCont.text,genderCont!,
                                birthCont.text, countryCont.text,cityCont.text];
                              json_user = jsonEncode(userForm);
                              addUser(json_user);
                              ShowDialog(context);
                              QrPainter QrImage = QrPainter(
                                data: "${nameCont.text}|${phoneCont.text}|${emailCont.text}",
                                version: QrVersions.auto,
                                errorCorrectionLevel: QrErrorCorrectLevel.Q,
                              );
                              saveQrImage(QrImage);
                              Future.delayed(const Duration(seconds: 2),(){
                                Navigator.pushAndRemoveUntil
                                  (context, MaterialPageRoute(
                                    builder: (context) => ChooseDevice()),(route) => false);
                              } );
                            } else{
                              // form is not submitted till all cells are filled
                            }
                          },
                          child: const Text("Sign up",
                            style: TextStyle(
                                fontSize: 18,
                              color: Color.fromRGBO(23, 129, 215, 0.5)
                            ),),
                        ))
                  ],
                )
            )
      )
    ]
    ),
    )
    );
}
```

Figure 4.55 sign-up build method

The figure below shows the UI for the sign-up page.

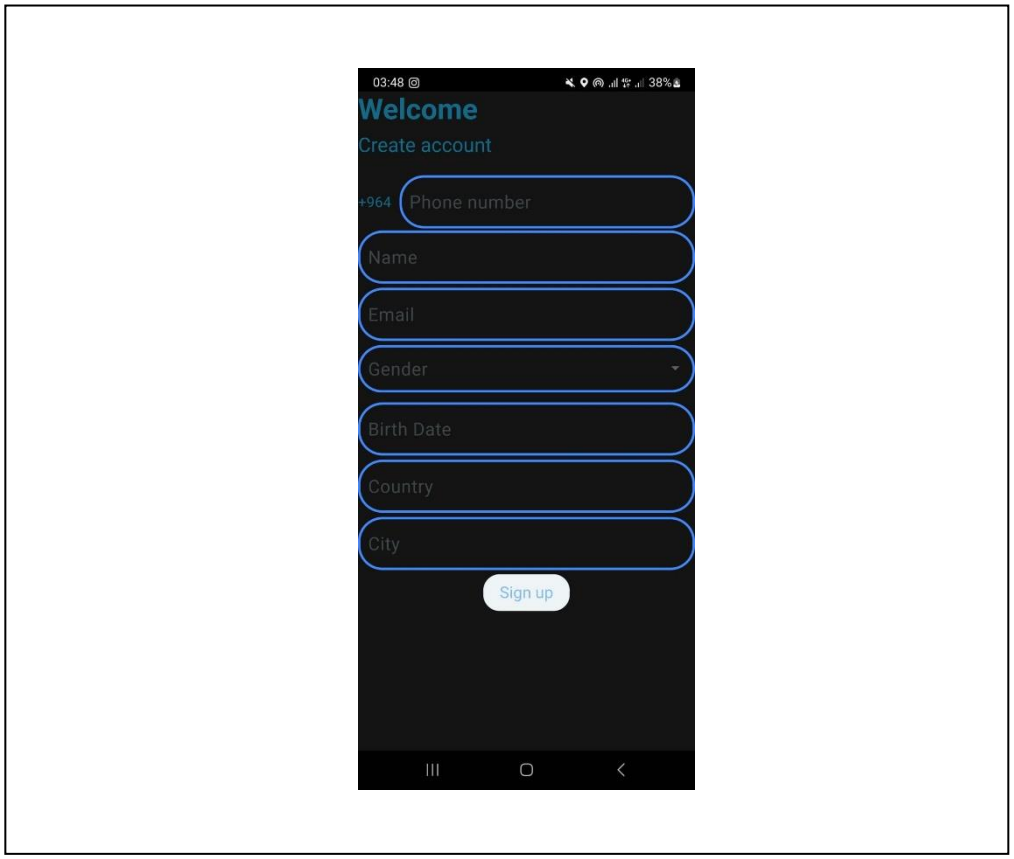
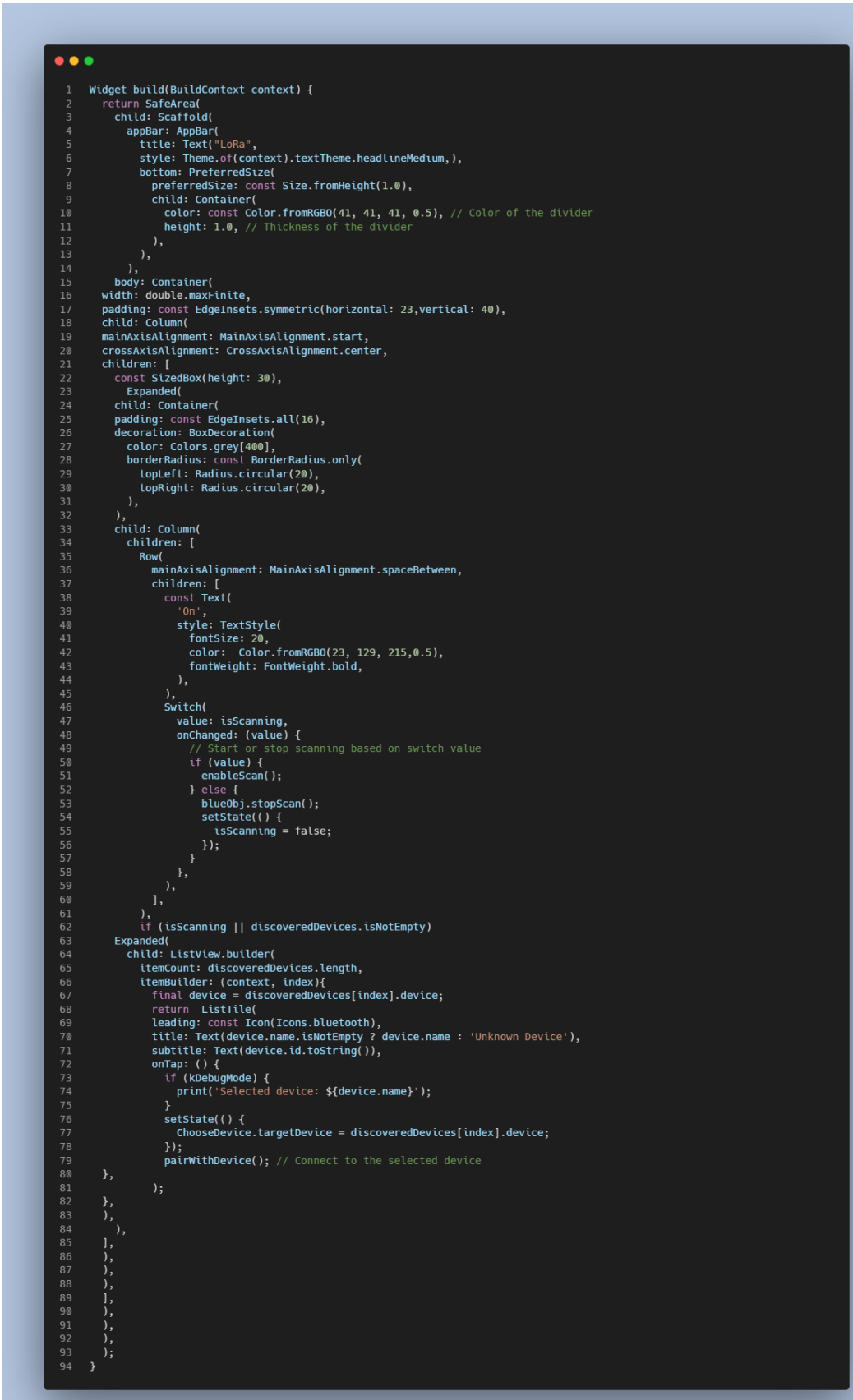


Figure 4.56 sign up page UI

After the user successfully signs up, the user is created, and they are navigated to the Bluetooth connect page. This page has a button for starting the Bluetooth device scanning. Once the button is pressed a list of scanned devices is displayed. The figure below shows the build method for the Bluetooth connect page.



```
Widget build(BuildContext context) {
  return SafeArea(
    child: Scaffold(
      appBar: AppBar(
        title: Text("LoRa",
        style: Theme.of(context).textTheme.headlineMedium,),
        bottom: PreferredSize(
          preferredSize: const Size.fromHeight(1.0),
          child: Container(
            color: const Color.fromRGBO(41, 41, 41, 0.5), // Color of the divider
            height: 1.0, // Thickness of the divider
          ),
        ),
      ),
    body: Container(
  width: double.maxFinite,
  padding: const EdgeInsets.symmetric(horizontal: 23,vertical: 40),
  child: Column(
  mainAxisAlignment: MainAxisAlignment.start,
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    const SizedBox(height: 30),
      Expanded(
    child: Container(
    padding: const EdgeInsets.all(16),
    decoration: BoxDecoration(
      color: Colors.grey[400],
      borderRadius: const BorderRadius.only(
        topLeft: Radius.circular(20),
        topRight: Radius.circular(20),
      ),
    ),
    child: Column(
      children: [
        Row(
          mainAxisAlignment: MainAxisAlignment.spaceBetween,
          children: [
            const Text(
              'On',
              style: TextStyle(
                fontSize: 20,
                color:  Color.fromRGBO(23, 129, 215,0.5),
                fontWeight: FontWeight.bold,
              ),
            ),
            Switch(
              value: isScanning,
              onChanged: (value) {
                // Start or stop scanning based on switch value
                if (value) {
                  enableScan();
                } else {
                  blueObj.stopScan();
                  setState(() {
                    isScanning = false;
                  });
                }
              },
            ),
          ],
        ),
        if (isScanning || discoveredDevices.isNotEmpty)
    Expanded(
      child: ListView.builder(
        itemCount: discoveredDevices.length,
        itemBuilder: (context, index){
          final device = discoveredDevices[index].device;
          return  ListTile(
          leading: const Icon(Icons.bluetooth),
          title: Text(device.name.isNotEmpty ? device.name : 'Unknown Device'),
          subtitle: Text(device.id.toString()),
          onTap: () {
            if (kDebugMode) {
              print('Selected device: ${device.name}');
            }
            setState(() {
              ChooseDevice.targetDevice = discoveredDevices[index].device;
            });
            pairWithDevice(); // Connect to the selected device
  },
          );
  },
  ),
    ),
  ],
  ),
  ),
  ),
  ],
  ),
  ),
  ),
  );
}
```

Figure 4.57 choose device build method

The figure below shows the UI design for the Bluetooth connect page.

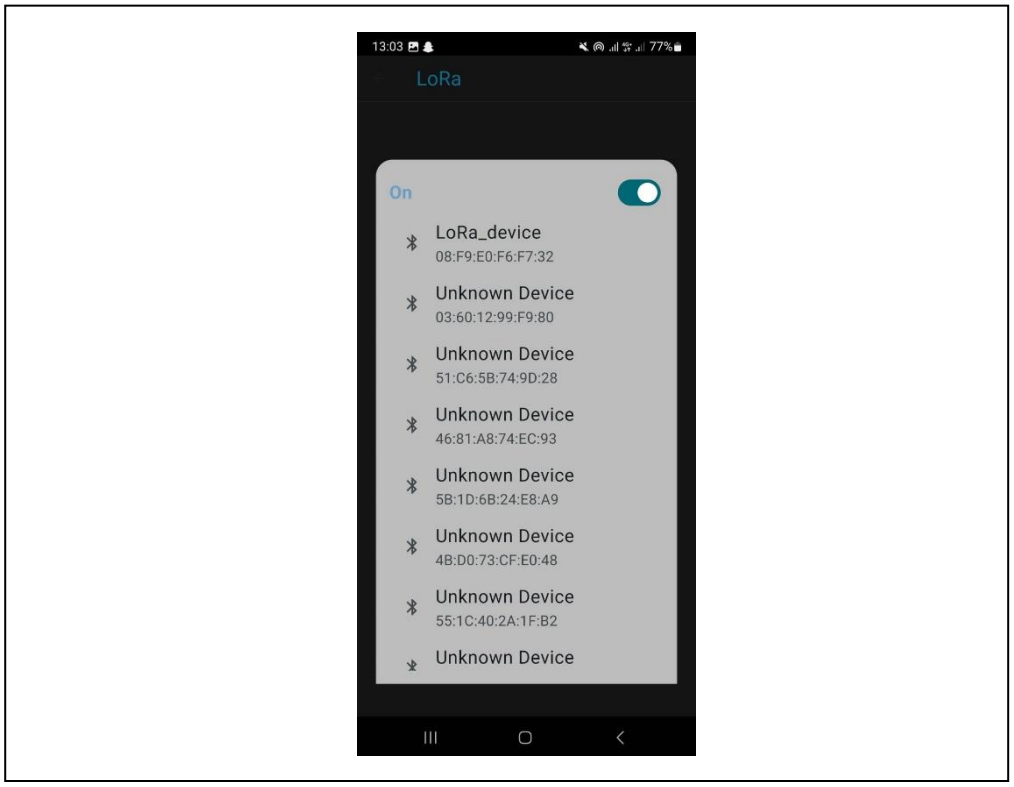
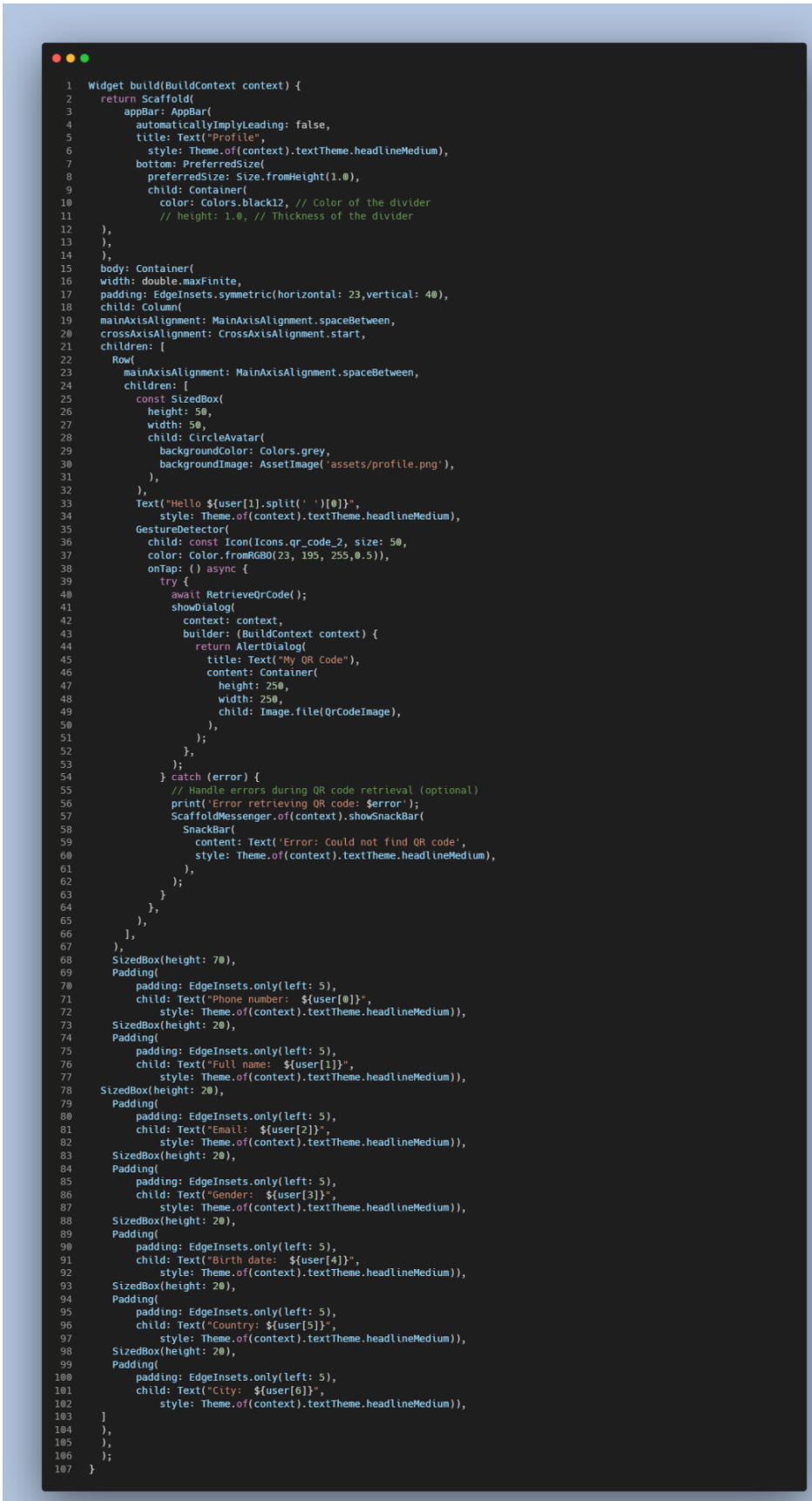


Figure 4.58 Bluetooth connect page UI

Once the desired device is connected, the user is navigated to the main page of the application which is the profile page. This page is the first page of a navigation bar made of three pages: profile, contact, and chat. This page displays all the information related to the user's account. The figures shown below display the navigation bar code and the code for the profile page interface.

```
Widget build(BuildContext context) {
  return  Scaffold(
    body: frames[currentIndex],
    bottomNavigationBar: BottomNavigationBar(
      currentIndex: currentIndex,
      onTap: (index) {
        setState(() {
          currentIndex = index;
        });
      },
        items: const [
          BottomNavigationBarItem(
              icon: Icon(Icons.chat),
              label: "Chat"),
          BottomNavigationBarItem(
              icon: Icon(Icons.account_circle),
              label: "Contacts"),
          BottomNavigationBarItem(
              icon: Icon(Icons.contacts),
              label: "Profile"),
        ],
    ),
  );
}
```

Figure 4.59 navigation build method



```dart
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        automaticallyImplyLeading: false,
        title: Text("Profile",
          style: Theme.of(context).textTheme.headlineMedium),
        bottom: PreferredSize(
          preferredSize: Size.fromHeight(1.0),
          child: Container(
            color: Colors.black12, // Color of the divider
            // height: 1.0, // Thickness of the divider
  ),
  ),
  ),
  body: Container(
  width: double.maxFinite,
  padding: EdgeInsets.symmetric(horizontal: 23,vertical: 40),
  child: Column(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        const SizedBox(
          height: 50,
          width: 50,
          child: CircleAvatar(
            backgroundColor: Colors.grey,
            backgroundImage: AssetImage('assets/profile.png'),
          ),
        ),
        Text("Hello ${user[1].split(' ')[0]}",
            style: Theme.of(context).textTheme.headlineMedium),
        GestureDetector(
          child: const Icon(Icons.qr_code_2, size: 50,
          color: Color.fromRGBO(23, 195, 255,0.5)),
          onTap: () async {
            try {
              await RetrieveQrCode();
              showDialog(
                context: context,
                builder: (BuildContext context) {
                  return AlertDialog(
                    title: Text("My QR Code"),
                    content: Container(
                      height: 250,
                      width: 250,
                      child: Image.file(QrCodeImage),
                    ),
                  );
                },
              );
            } catch (error) {
              // Handle errors during QR code retrieval (optional)
              print('Error retrieving QR code: $error');
              ScaffoldMessenger.of(context).showSnackBar(
                SnackBar(
                  content: Text('Error: Could not find QR code',
                  style: Theme.of(context).textTheme.headlineMedium),
                ),
              );
            }
          },
        ),
      ],
    ),
    SizedBox(height: 70),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Phone number:  ${user[0]}",
            style: Theme.of(context).textTheme.headlineMedium)),
    SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Full name:  ${user[1]}",
            style: Theme.of(context).textTheme.headlineMedium)),
  SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Email:  ${user[2]}",
            style: Theme.of(context).textTheme.headlineMedium)),
    SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Gender:  ${user[3]}",
            style: Theme.of(context).textTheme.headlineMedium)),
    SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Birth date:  ${user[4]}",
            style: Theme.of(context).textTheme.headlineMedium)),
    SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("Country: ${user[5]}",
            style: Theme.of(context).textTheme.headlineMedium)),
    SizedBox(height: 20),
    Padding(
        padding: EdgeInsets.only(left: 5),
        child: Text("City:  ${user[6]}",
            style: Theme.of(context).textTheme.headlineMedium)),
  ]
  ),
  ),
  );
}
```

Figure 4.60 profile frame build method

The figure below illustrates the profile page UI design.

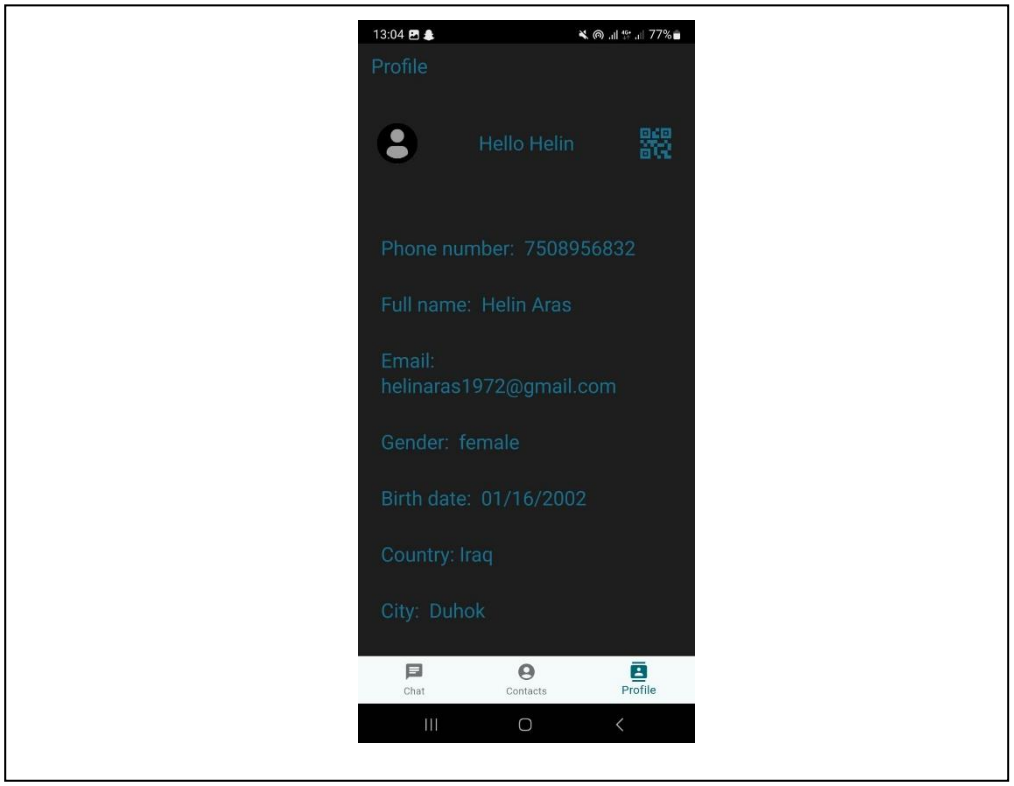


Figure 4.61 profile page UI

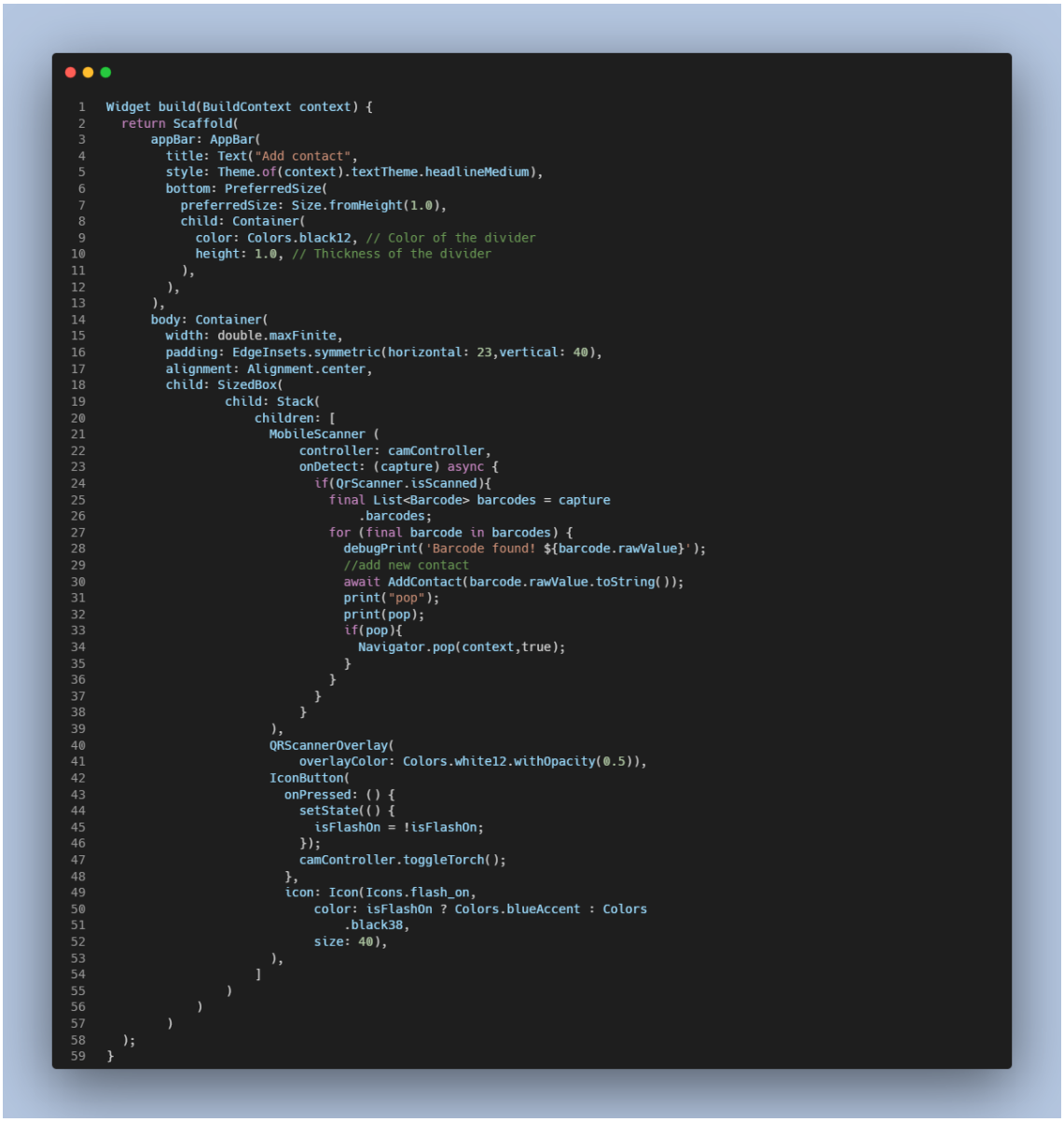
The user may go to the contacts page where list of added contacts. inside the contacts page, there is a button named “Add contact” that navigates to another page called the QR scanner. This page scans the QR code for other users and adds them to the contact list. The figures depicted below show the codes for the contact page (figure 4.62) and QR scanner page (figure 4.63).

```
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        automaticallyImplyLeading: false,
      title: Text("Contacts",
          style: Theme.of(context).textTheme.headlineMedium),
  bottom: PreferredSize(
  preferredSize: Size.fromHeight(1.0),
  child: Container(
  color: Colors.black12, // Color of the divider
  height: 1.0, // Thickness of the divider
  ),
  ),
  ),
  floatingActionButton: FloatingActionButton.extended(
      onPressed: () async {
        await Navigator.push(context, MaterialPageRoute(
            builder: (context) => QrScanner()));
        RetrieveContactList();
      },
      label: const Text("Add Contact"),
  icon: Icon(IconlyBroken.add_user),),
  body: Container(
  width: double.maxFinite,
  padding: EdgeInsets.symmetric(vertical: 20),
    child: QrScanner.contacts.isEmpty ? Center(child: Text("No contacts available"))
    : ListView.separated(
      itemCount: QrScanner.contacts.length,
        itemBuilder: (BuildContext context, int index) {
        final contact = QrScanner.contacts[index];
        return ListTile(
          horizontalTitleGap: 10,
          title: Text(contact.name, style: Theme.of(context).textTheme.bodyMedium),
          subtitle: Text("${contact.phone}     ${contact.email}",
          style: Theme.of(context).textTheme.bodySmall),
          leading: CircleAvatar(
            backgroundColor: Colors.grey,
            backgroundImage: AssetImage('assets/profile.png'),

          ),
          trailing:  Row(
            mainAxisSize: MainAxisSize.min,
            children: [
              IconButton(onPressed: (){
                setState(() {
                  QrScanner.contacts.removeAt(index);
                  QrScanner.chats.removeAt(index);
                });
              },
              icon: Icon(IconlyBroken.delete),
              ),
            ],
        )
        );
        },
        separatorBuilder: (BuildContext context, int index) => const Divider(
          color: Colors.white,
        )),
  )
  );
}
```

Figure 4.62 contact frame build method

As the code depicts, a floating action button is created that navigates to the QR scanner page. When the new user is scanned and saved, the QR scanner page is automatically popped and navigates back to the contact page where the updated contact list is displayed. Correspondingly the chat list is updated as well.



```
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        title: Text("Add contact",
        style: Theme.of(context).textTheme.headlineMedium),
        bottom: PreferredSize(
          preferredSize: Size.fromHeight(1.0),
          child: Container(
            color: Colors.black12, // Color of the divider
            height: 1.0, // Thickness of the divider
          ),
        ),
      ),
      body: Container(
        width: double.maxFinite,
        padding: EdgeInsets.symmetric(horizontal: 23,vertical: 40),
        alignment: Alignment.center,
        child: SizedBox(
                child: Stack(
                    children: [
                      MobileScanner (
                          controller: camController,
                          onDetect: (capture) async {
                            if(QrScanner.isScanned){
                              final List<Barcode> barcodes = capture
                                  .barcodes;
                              for (final barcode in barcodes) {
                                debugPrint('Barcode found! ${barcode.rawValue}');
                                //add new contact
                                await AddContact(barcode.rawValue.toString());
                                print("pop");
                                print(pop);
                                if(pop){
                                  Navigator.pop(context,true);
                                }
                              }
                            }
                          }
                      ),
                      QRScannerOverlay(
                          overlayColor: Colors.white12.withOpacity(0.5)),
                      IconButton(
                        onPressed: () {
                          setState(() {
                            isFlashOn = !isFlashOn;
                          });
                          camController.toggleTorch();
                        },
                        icon: Icon(Icons.flash_on,
                            color: isFlashOn ? Colors.blueAccent : Colors
                                .black38,
                            size: 40),
                      ),
                    ]
                )
            )
        )
      )
  );
}
```

Figure 4.63 QR scanner build method

This is the code for the scanner where the *MobileScanner* widget is created to access the camera and read the QR code. When the barcode is found, a dialog is displayed to show the user's data that was retrieved from the QR code, and only the name field can be edited. The user can save the user by pressing the *save* button in the dialog. The illustrations below display the UI design for the contact page and QR scanner page.

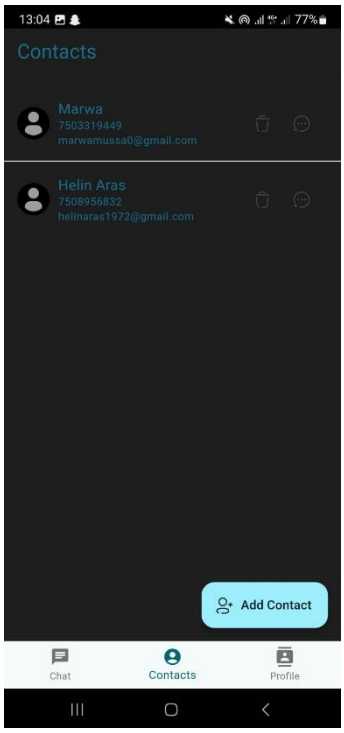


Figure 4.64 Contact page UI

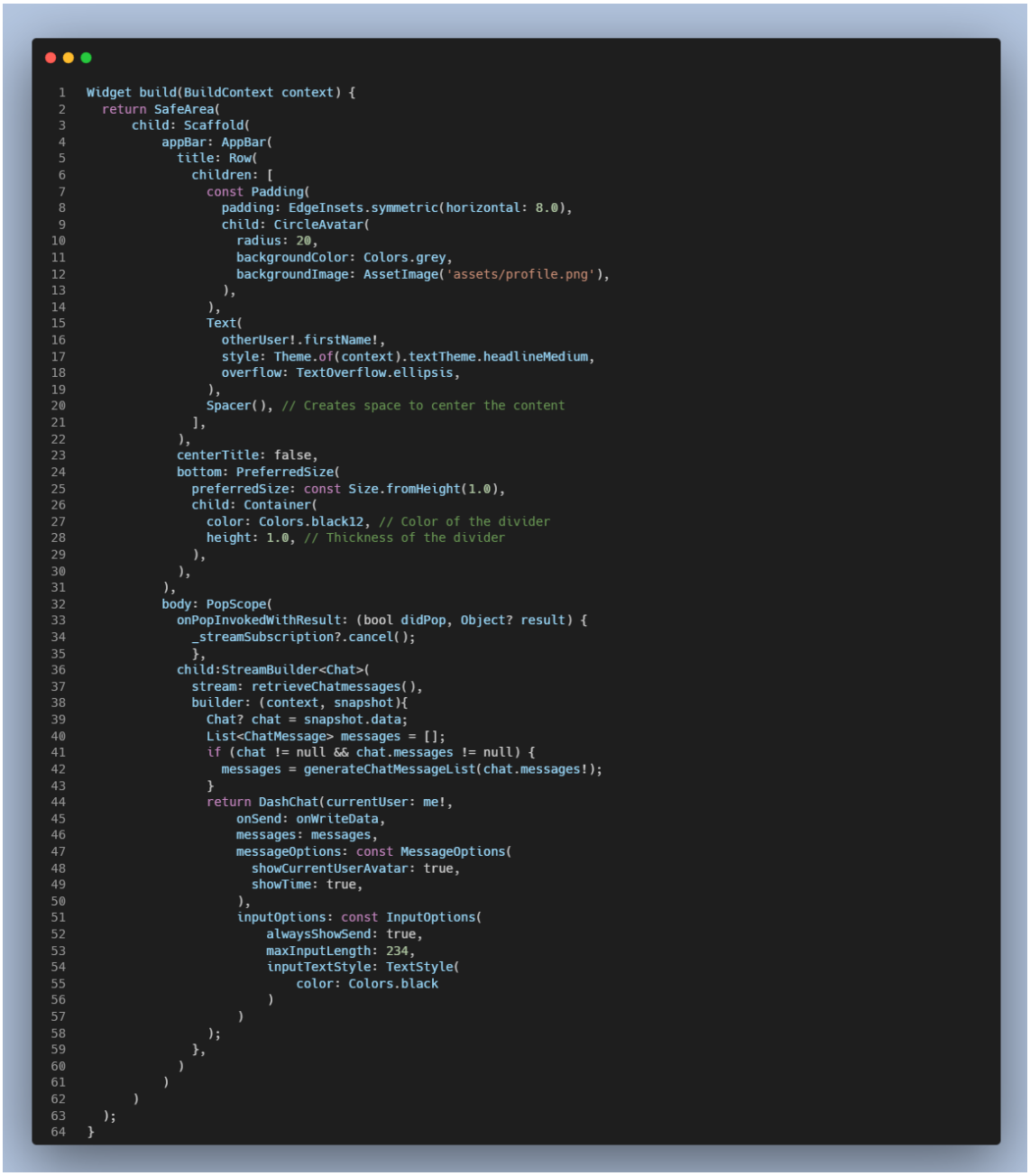
Figure 4.65 QR scanner page UI

Next, if the user wishes to chat with one of their contacts, they can navigate to the chat page where the list of chats is displayed. When a chat is pressed, and the session is created successfully, the user is navigated to the chat box where they can send messages and receive them. Messages are printed on the screen in the chat box style. The figures below display the codes for the chat page screen and chat box screen.

```
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(...),
      body: Container(
        width: double.maxFinite,
        padding: const EdgeInsets.symmetric(vertical: 20),
        child: ListView.separated(
            itemCount: QrScanner.chats.length,
            itemBuilder: (BuildContext context, int index) {
              final chat = QrScanner.chats[index];
              return ListTile(
                onTap: (){...},
                  title: Text(chat.participants![0].name, style: Theme.of(context).textTheme.bodyMedium),
                  subtitle: Text("say Hi! to ${chat.participants![0].name}",
                      overflow: TextOverflow.ellipsis,
                      style: const TextStyle(
                      fontSize: 11,
                      letterSpacing: -0.2,
              fontWeight: FontWeight.w600,
              color: Color(0xFF9899A5),
                      )),
                  leading: const CircleAvatar(
                    backgroundColor: Colors.grey,
                    backgroundImage: AssetImage('assets/profile.png'),
                  ),
              );
            },
            separatorBuilder: (BuildContext context, int index) => const Divider(
              color: Colors.white,
            )),
      )
  );
}
```

Figure 4.66 chat frame build method

The code in the figure below is the build method for the chat box. The build method creates a stream builder that gets the stream of messages from the *retrieveChatMessages* method, and it returns a widget called *DashChat* from the dash_chat_2 package. In this widget, the method *onWriteData* is called when the send button is pressed on the screen. Also, notice how the number of allowed characters per message is limited to 234 in *inputOption* in *DashChat.* This limitation is applied because each data packet in our system has a payload size of 234 bytes, and therefore the user is prevented from entering more than that limit per message sent.



```dart
Widget build(BuildContext context) {
  return SafeArea(
      child: Scaffold(
          appBar: AppBar(
            title: Row(
              children: [
                const Padding(
                  padding: EdgeInsets.symmetric(horizontal: 8.0),
                  child: CircleAvatar(
                    radius: 20,
                    backgroundColor: Colors.grey,
                    backgroundImage: AssetImage('assets/profile.png'),
                  ),
                ),
                Text(
                  otherUser!.firstName!,
                  style: Theme.of(context).textTheme.headlineMedium,
                  overflow: TextOverflow.ellipsis,
                ),
                Spacer(), // Creates space to center the content
              ],
            ),
            centerTitle: false,
            bottom: PreferredSize(
              preferredSize: const Size.fromHeight(1.0),
              child: Container(
                color: Colors.black12, // Color of the divider
                height: 1.0, // Thickness of the divider
              ),
            ),
          ),
          body: PopScope(
            onPopInvokedWithResult: (bool didPop, Object? result) {
              _streamSubscription?.cancel();
              },
            child:StreamBuilder<Chat>(
              stream: retrieveChatmessages(),
              builder: (context, snapshot){
                Chat? chat = snapshot.data;
                List<ChatMessage> messages = [];
                if (chat != null && chat.messages != null) {
                  messages = generateChatMessageList(chat.messages!);
                }
                return DashChat(currentUser: me!,
                    onSend: onWriteData,
                    messages: messages,
                    messageOptions: const MessageOptions(
                      showCurrentUserAvatar: true,
                      showTime: true,
                    ),
                    inputOptions: const InputOptions(
                        alwaysShowSend: true,
                        maxInputLength: 234,
                        inputTextStyle: TextStyle(
                            color: Colors.black
                        )
                    )
                );
              },
            )
          )
      )
  );
}
```

Figure 4.67 chat box build method

The two figures below depict the UI design for the chat page and chat box screen.

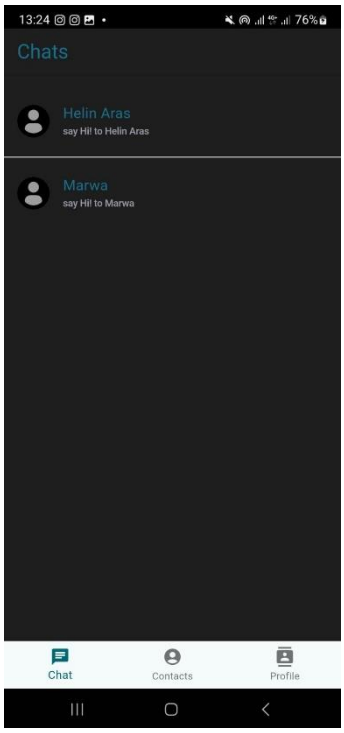


Figure 4.68 chat page UI

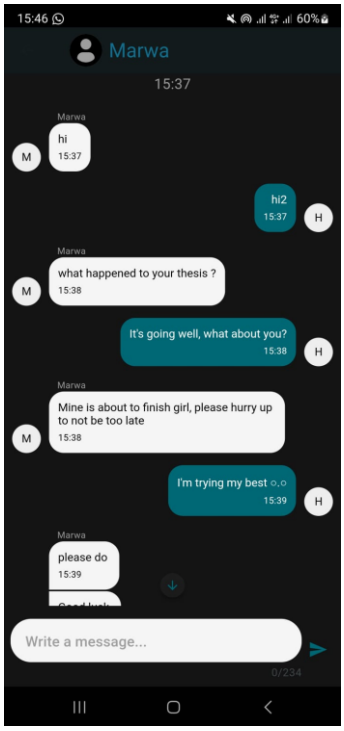


Figure 4.69 chat box UI

Alert dialogs

The codes for all the alert dialogs in this application were displayed and demonstrated in the backend section. In this part of the frontend section, all those alert dialogs UI designs are displayed in the figures below.

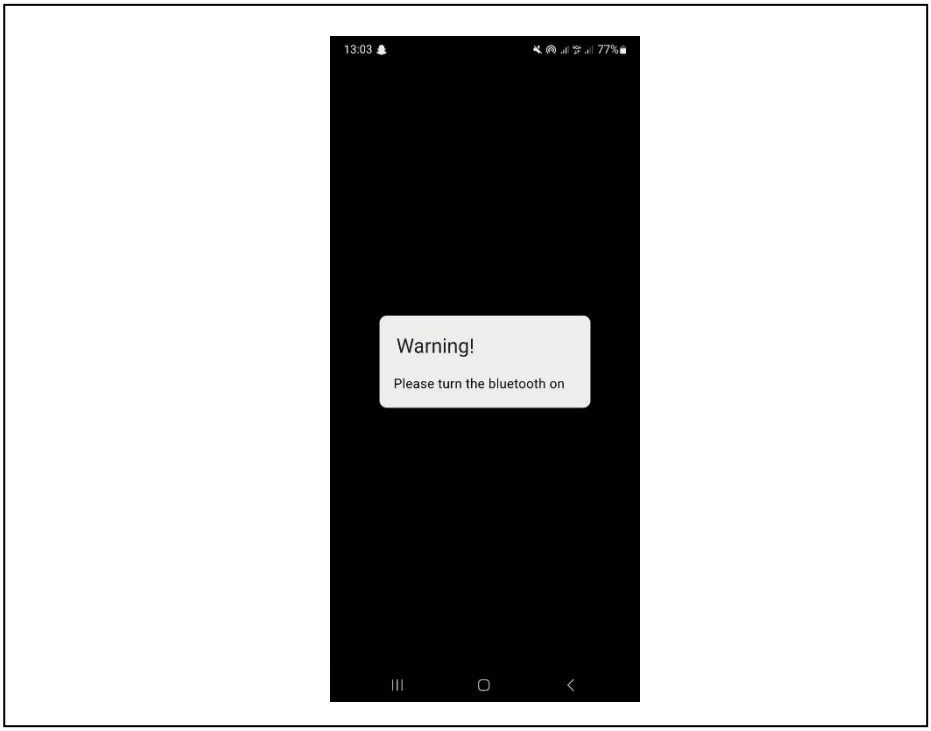


Figure 4.70 Bluetooth state alert dialog

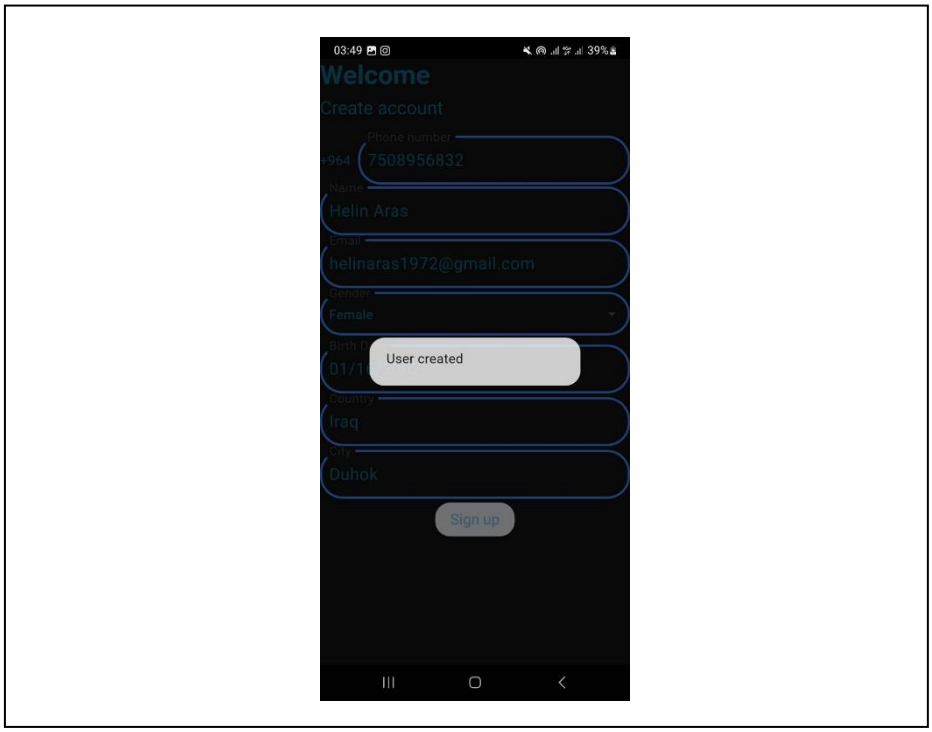


Figure 4.71 User sign-up alert dialog

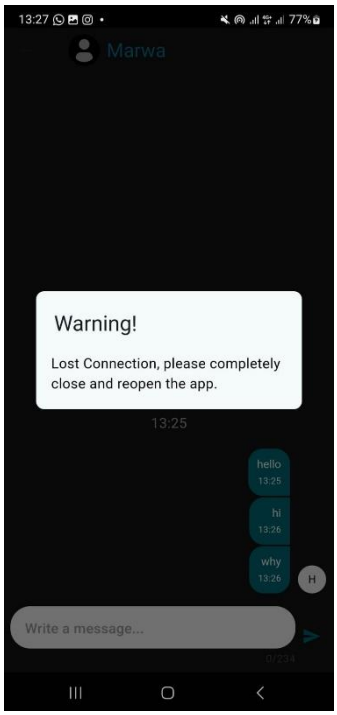


Figure 4.72 BLE connection interrupt alert dialog

Figure 4.73 QR code preview alert dialog

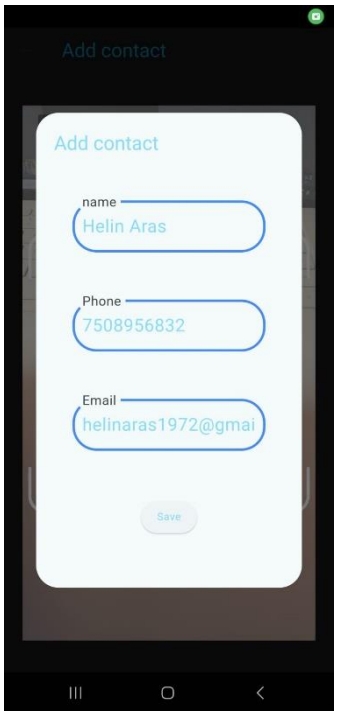


Figure 4.74 add contact alert dialog

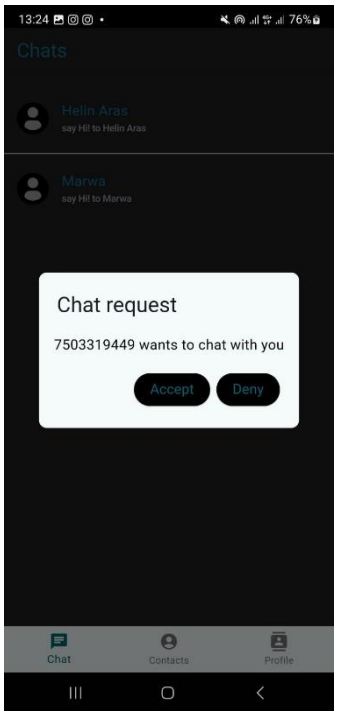


Figure 4.75 chat request alert dialog

Chapter 5 Testing and Results

This chapter is complementary to the implementation chapter. In the implementation chapter, we monitored the operation of the building blocks of our project, in this chapter, we will portray the tests that this project underwent to produce its functionalities. Similar to the topic distribution in the previous chapter, this chapter is divided into sections based on the topics covered in the project which are: hardware testing, network and security testing, and mobile application testing. Furthermore, for every element of this project, the test context, errors, the methodology used to resolve the issues found, and the obtained result of this process are examined.

5.1 Hardware Testing

The hardware device used for this project is the TTGO T-Beam ESP32-based board. As previously mentioned, we have utilized the LoRa and BLE network on this board to communicate with other devices in the network created for the project. The requirements, trials, and the process of formulating the LoRa and BLE are inspected in this section.

5.1.1 LoRa testing

- Testing LoRa functionality: Initially, we tested the connection by the two TTGO devices by sending a counter value. The figure below displays two LoRa devices where one is sending data, and the other is receiving data.

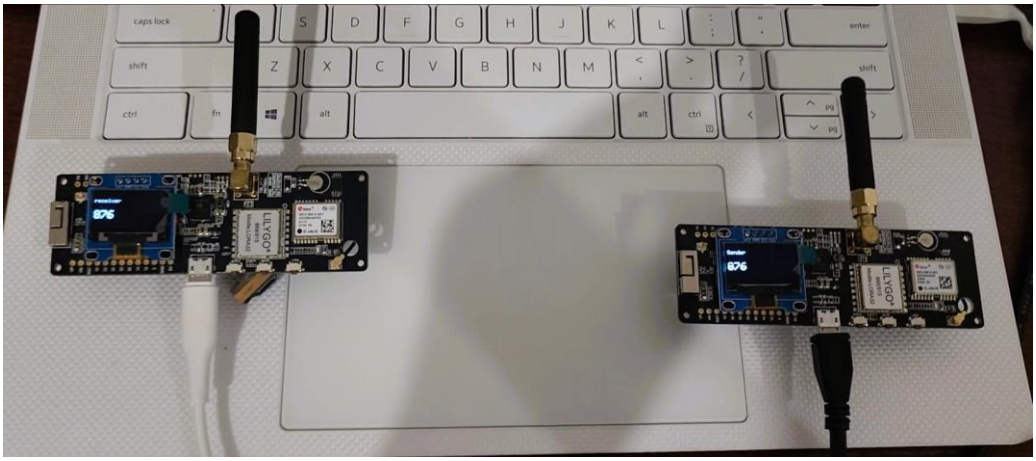


Figure 5.1 LoRa transceivers

- Distance measurement: testing the maximum distance the LoRa can transmit data.

- Packet size: measuring the packet size effects on the packet delay and maximum distance the packet can travel.

To test the LoRa modulation's distance travel capability and the data packet's effect on this subject, we conducted a test on the LoRa connection on the device. Initially, a test was implemented to perceive the maximum distance that the LoRa device can send a stable signal and the delay progress with the distance. Hence, a test was attempted to measure how far can our LoRa devices send and receive signals without distortion, in an urban non-line-of-sight area, with obstacles. The test was conducted in the Sami Abdulrahman park in the capital Erbil. Although the limits of LoRa signal transmission are specified by the manufacturer, however, the test was applied to get more realistic results, and eventually, the results did have some variance from the estimated values. The test was applied in two scenarios: one where the data packet size is the maximum (255 bytes), and in the other the data packet size is the minimum (1 byte). The table below shows the results of our test.

Table 5.1 LoRa measurements

Packet size	**Stable range**	**Unstable range**	**Delay in stable range**	**Delay in unstable range**
255 bytes	0 – 400 m	400 – 500 m	2s	29s
1 byte	0 – 350 m	350 – 450 m	0.3s	1.1s

Even though there was no test done for rural areas in this project, It is crucial to note that in rural areas, and especially at line-of-sight distance, the range of LoRa is perceived to be greater than the results obtained from an urban non-line-of-sight area.

5.1.2 Bluetooth LE testing

- BLE Characteristics: determine how to define the characteristics and services of the BLE in the TTGO T-Beam

- BLE properties: assign properties to BLE characteristics

BLE is another functionality in the TTGO module that we used for this project. The BLE's role in our hardware device is to make a connection with the BLE on the mobile phone to send and receive messages. The BLE devices consist of services that hold characteristic values and are assigned to properties that determine their usage.

Initially, in my project, I created one characteristic for reading, writing and notifying, data. However, this turned out to be inefficient because the BLE could not receive and send data at the same time. To fix this, I created two different characteristics in my service, each assigned to the write property which receives data, and the other assigned to the read and notify property which sends data and notifies the end user when new data is entered. After this edit, the data could send and receive data simultaneously without any issue.

It's also important to note that the "notify" property is essential to assign to the characteristic that sends data to the peripheral (in this case the mobile phone). This property notifies the receiver each time new data is received by the sender.

5.2 Network and Security Testing

5.2.1 Networking Testing

- Data packet size: determining the MTU for the data packet
- Data packet design: designing the header and payload

Following the configuration of the LoRa and BLE service on our device, we have configured a network protocol for the data we work with. As defined by the manufacturer, the maximum data packet size in LoRa is 255 bytes which is why we are bound to arrange a data packet of this size in our network for the LoRa and BLE. The first thing we did was to design a data packet that was divided into two main parts: the payload which contains the data we want to transmit, and the header,

which has all the metadata about the payload.

The first data added to the header were unique identifiers for both the sender and the receiver. For this, we have assigned a 10-byte long identifier that is fetched from the user's account on the mobile application. In total, the identifiers of both users occupy 20 bytes of the header. After testing and ensuring that data was recognized by the users, we faced another issue which was knowing the content type for the data in the payload. The type of the payload can differ from exchanging key attributes to sending data or responding to data. For this purpose, we have added an extra byte to identify the type of data that is transmitted. In total, the header size is 21 bytes, and the payload is 234 bytes. The logic of this network is all coded in the devices of this network and has proven to successfully communicate together. The same logic was applied to the network between LoRa users and the network between BLE users. It's crucial to note that when you read the data from using BLE from the mobile application to the TTGO T-Beam, the packet is received as a string in the callback funtion, and therefore you need to move your data to a byte array in order to read the data byte by byte to inspect the header and payload.

5.2.2 Security Testing

- Choosing the most ideal encryption algorithm for our board
- Creating a key generation method suitable for the encryption algorithm

The LoRa module in the T-Beam board is a physical layer modulation technology and therefore does not provide security measurements while the BLE is secured using advanced encryption algorithms and other security measures. Therefore, an encryption algorithm was critical to add to the LoRa communication network. Initially, we applied the AES algorithm to our data in the T-Beam. However, we faced a storage error because the size of the program with AES encryption was greater than our board's capacity, thus applying AES was impractical. The figure below displays the text for this error.

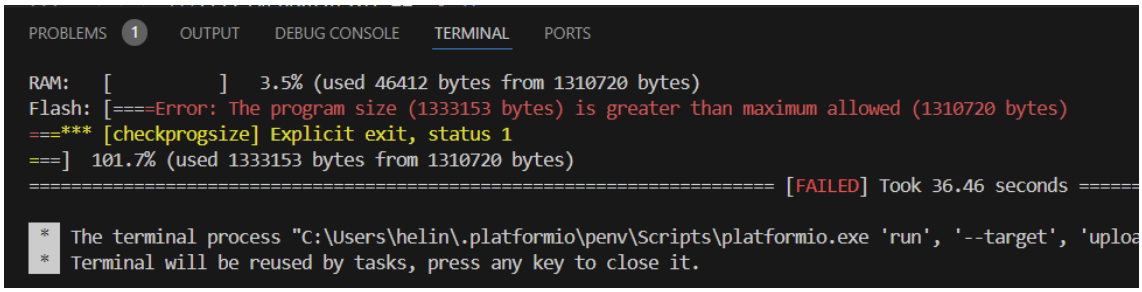


Figure 5.2 program size error

Therefore, a simpler, stronger encryption method had to be used. For this, one-time pad algorithm was chosen. The One-time pad is simple in its calculations and thus doesn't take much space, yet it is known as the unbreakable encryption. However, there was one problem with this algorithm, and that was the key. The key size must be at least the same size as the data, and it must be a random value. To fix this error, we created a method for generating a 256-byte key, and we used the timestamps of both users as parameters for this method. The reason for this is that the timestamps are completely random and non-repetitive values, and thus they create a random key. Moreover, the method takes the timestamps, each of which is 16 bytes, and creates a 256-byte key that is greater than the MTU of our network. The key transmission issue is solved as well, by exchanging timestamps between users instead of the actual key. Once the users have the timestamps, they can input them into the methods, and the same key will output on both sides. These steps are completed before any data can be sent. Hence, once keys are created, the session is created, and devices are allowed to exchange data using the key. It's important to note that the session is terminated each time one of the users closes the chat interface, and when the user enters the chat interface again, another session is created with another session key.

5.3 Mobile Application Testing

The mobile application development is a crucial phase of this project. It is the joining point between the user and the network system. Initially, a test application was created that solely does the BLE connection with the T-Beam, reading and writing data. This draft application was developed to ensure the BLE connection is functioning properly. The figures below display the draft application.

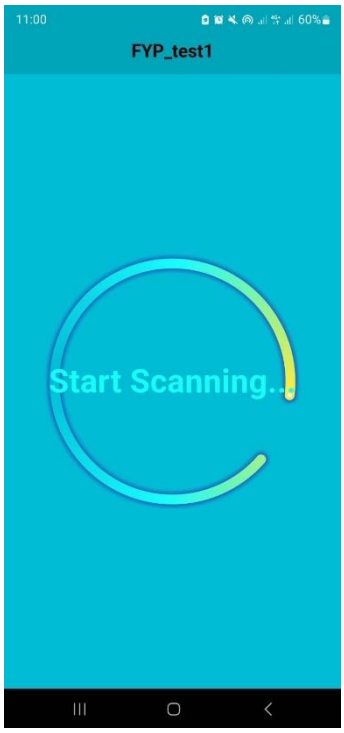


Figure 5.3 scanning for BLE devices

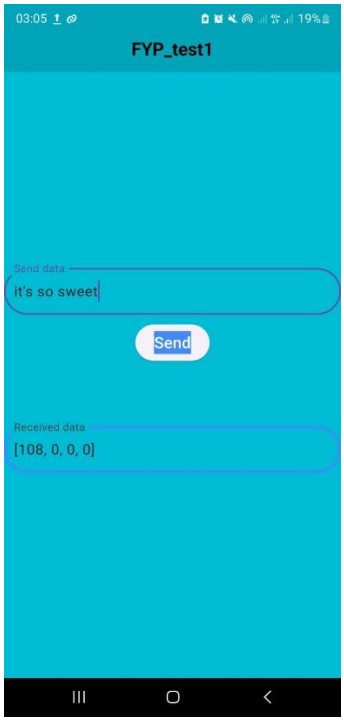


Figure 5.4 data send and receive

Eventually, using the draft application, the first successful test for sending and receiving data was coordinated. The images below show the communication between two mobile phones using the TTGO T-Beam as a medium.

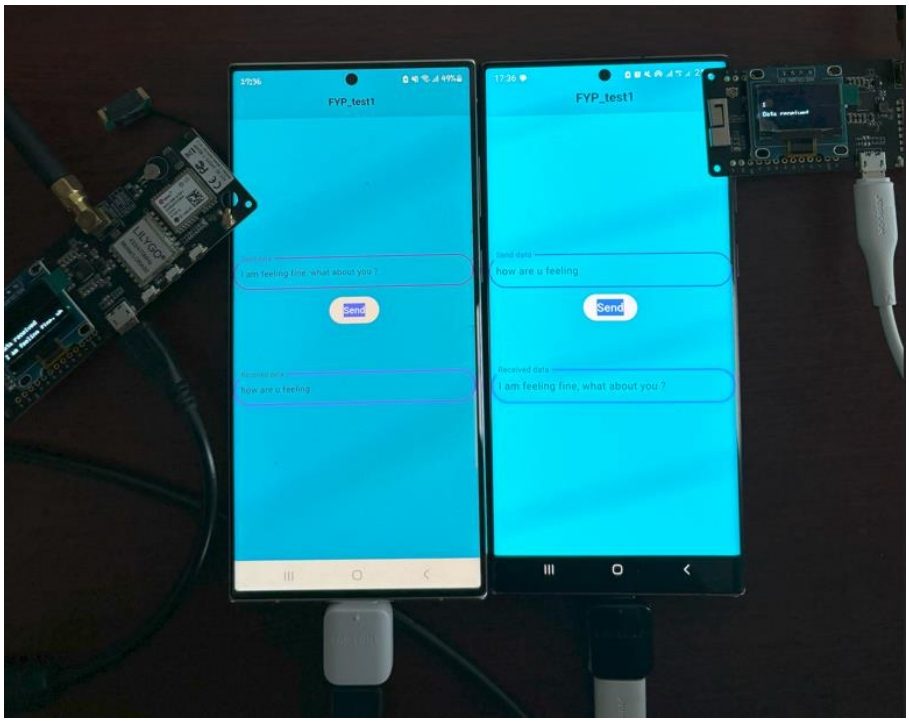


Figure 5.5 connection established between two mobile phones

Following this process, the test application was developed into another fully functional application with the additional features required for the system acquired in this project. The new application contains user accounts, chat lists, contacts, contact lists, QR codes for user accounts, QR scanning to add contacts, and a local database to save all the data related to the user's account, contacts, and chat messages. The testing for this project's application is entirely manually done. The process goes as follows: the entire application is traversed page by page, the features within each page are tested, and lastly, the tests for each page are organized in separate tables.

- Page 1: the starting page of the application.

Table 5.2 page 1 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	The BLE button is OFF	When opening the application	The alert dialog should display	Alert dialog is displayed	Pass
TC02	The BLE button is ON	When opening the application	The start page should be displayed.	The start page is displayed	Pass
TC03	The user account is signed up	When pressing the start button	The user should be navigated to the BLE connect page	The user is navigated to the BLE connect page	Pass
TC04	The user is not signed up yet	When pressing the start button	The user should be navigated to the sign-up page	The user is navigated to the sign-up page	Pass

- Page 2: the sign-up page (assuming the user is not registered).

Table 5.3 page 2 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	If all col-	When filling in	The user	The user is	Pass

	umns are filled and the phone number is correct	the fields to sign-up	should successfully sign up and go to the BLE connect page	registered	
TC02	If all columns are not filled	When filling in the fields to sign-up	The validators should prevent the user from signing up	The user is not registered	Pass
TC03	If the phone number is not 10 digits	When filling in the fields to sign-up	The validators should prevent the user from signing up	The user is not registered	Pass
TC04	If the phone number is 10 digits and other columns are filled correctly	When filling in the fields to sign-up	The user should successfully sign up and go to the BLE connect page	The user is registered	Pass
TC05	The phone number field should	When the phone field is entered with values	The user should only be allowed to enter numbers	The user is only allowed to enter numbers.	Pass

	only allow number entry				
TC06	Navigation to the sign-up field	user navigation to the sign-up page after being registered	The user shouldn't be able to navigate to the sign-up page after being registered	The user can't navigate to the sign-up page after being registered	Pass
TC07	The phone field	When the phone number is entered	The phone field should take all country codes and adjust the identifier size accordingly	The phone field takes only Iraq country codes	Fail

- Page 3: Bluetooth available devices.

Table 5.4 page 3 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	The button is OFF	When tapping the button	The application should not scan for devices	The application does not scan for devices	Pass
TC02	The button is ON	When tapping the button	The application should	The list of connected	Pass

			scan for devices	devices is shown	
TC03	Targeted TTGO device is clicked	After pressing the scan button	The device should be connected and navigates to the profile page	The device is connected and navigate to the profile page	Pass
TC04	Another device is clicked	After pressing the scan button	The device should not be able to connect and does not navigate to the profile page	The device is not connected and does not navigate to the profile page	Pass
TC05	Permission for nearby devices	When BLE is scanned for the first time on the application	The permission notification should appear	The permission notification appears	Pass
TC06	Permission for device location	When BLE is scanned for the first time on the application	The permission notification should appear	The permission notification appears	Pass
TC07	Bluetooth disconnection	When Bluetooth is disconnected at	An alert dialog should show telling	An alert dialog shows tell-	Pass

		any point in the app	the user to close the app and reopen to reconnect	ing the user to close the app and reopen to reconnect	

- Page 4: The profile page (initial page of the navigation bar).

Table 5.5 page 4 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	The QR code icon	When tapping on the QR code icon	A dialog should show displaying the user's QR code	A dialog is shown displaying user's QR code	Pass
TC02	User data displayed on the screen	When navigating to the profile page	The user's data should be printed on the profile page	The user's data is printed on the profile page	Pass
TC03	Navigation bar	When traversing through navigation bar indices	The user should be able to go to other pages of the navigation bar	The user can go to other pages of the navigation bar	Pass

- Page 5: The contact page (second page of the navigation bar)

Table 5.6 page 5 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	Navigation bar	When traversing through navigation bar indices	The user should be able to go to other pages of the navigation bar	The user can go to other pages of the navigation bar	Pass
TC02	The add contact button	When pressing the add contact button	The user should be able to navigate to the QR scanner page	The user can navigate to the QR scanner page	Pass
TC03	The contact list	When going to the contact page	The contact list should appear	The contact list appears	Pass
TC04	The delete icon	When pressing the delete icon of a contact	The contact and chat of the contact should be deleted	The contact and chat of the contact is deleted	Pass
TC05	The contact information	Data displayed for each contact	Data scanned for a contact should be displayed in the contact tile	Data scanned for a contact is displayed in the contact tile	Pass

TC06	The message icon	When the message icon is pressed	The user should be navigated to the chat when the message icon is pressed	The user can't navigate to the chat when the message icon is pressed	Fail

- Page 6: QR scanner page (scan a user)

Table 5.7 page 6 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	Navigation bar	When traversing through navigation bar indices	The user should be able to go to other pages of the navigation bar	The user can go to other pages of the navigation bar	Pass
TC02	Permission for taking pictures and recording videos	When the user presses the add contact button for the first time	The permission notification should appear	The permission notification appears	Pass
TC03	The flash icon	When the flash icon is pressed	The flash should turn on when the icon	The flash turns on when the	Pass

			is pressed	icon is pressed	
TC04	The camera scanner	When the camera is pointed to the QR code	The camera should scan for the QR code	The camera scans for the QR code	Pass
TC05	The QR code	When the QR code scanned is not in the targeted form	The QR scanner should not display the values of the QR code	The QR scanner does not display the values of the QR code	Pass
TC06	The QR code	When the QR code scanned is in the targeted form	The QR scanner should display the values of the QR code	The QR scanner displays the values of the QR code	Pass
TC07	The dialog to display scanned data	When the user does not exist in the contact list	The user should be able to save the scanned user in the contact list	The user can save the scanned user in the contact list	Pass
TC08	The dialog to display scanned	When the user exists in the contact list	The user should not be able to save the scanned user in	The user can’t save the scanned	Pass

	data		the contact list	user in the contact list	
TC09	Saving the user in the contact list	When pressing the save button in the scanned data dialog	The dialog frame and QR scanner page should be popped and the user should be navigated back to the contact page and the updated contact list should be displayed	The dialog frame and QR scanner page are popped, the user is navigated back to the contact page and the updated contact list is displayed	Pass

- Page 7: Chat page (third page of the navigation bar)

Table 5.8 page 7 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	Navigation bar	When traversing through navigation bar indices	The user should be able to go to other pages of the navigation bar	The user can go to other pages of the navigation bar	Pass
TC02	The chat list	When going to the chat page	The chat list should appear	The chat list appears	Pass

TC03	Chat tile	When pressing a chat	A chat request should be sent to the chat's other participant	A chat request is sent to the chat's other participant	Pass
TC04	Chat request reply	When a user sends a chat request and the reply to the request is accepted	The user should automatically be navigated to the chat box	The user is automatically navigated to the chat box	Pass
TC05	Chat request reply	When a user sends a chat request and the reply for the request is denied	The user should get a "request denied" alert dialog, and should not be able to go to the chat box	The user gets a "request denied" alert dialog, and can't go to the chat box	Pass
TC06	The chat request	Chat request dialog	When a chat request is received, a chat request dialog should show	When a chat request is received, the chat request dialog shows	Pass
TC07	Chat request	When the accept chat but-	The user should au-	The user is automati-	Pass

	dialog content	ton is pressed	tomatically be navigated to the chat box	cally navigated to the chat box	
TC08	Chat request dialog content	When the deny chat button is pressed	A request denied reply should be sent back, and the user shouldn't be able to go to the chat box	A request denied reply is sent back, and the user can't go to the chat box	Pass
TC09	Contact deleted	When a contact is deleted from the contact list	The chat entity for that contact should be deleted	The chat entity for that contact is deleted	Pass

- Page 8: the chat box

Table 5.9 page 8 test cases

Test ID	Description	Steps	Expected result	Actual result	Status
TC01	Previous chat	When entering a chat	The previous chats should be displayed	The previous chats are displayed	Pass
TC02	Messages order in the	How chats are displayed when text is	The messages should be displayed	The messages are displayed	Pass

	chat	sent or received	according to the time they are created at	according to the time they were created at	
TC03	Message length	When typing a message	The characters per message should be limited to 234 characters	The characters per message are limited to 234 characters	Pass
TC04	The send button	When pressing the send button	The message should be printed on the screen and sent to the receiver when the send button is pressed	The message is printed on the screen and sent to the receiver when the send button is pressed	Pass
TC05	Received messages	When a message is received	Received messages should be printed on the screen	Received messages are printed on the screen	Pass
TC06	Leaving a chat	When the user leaves a chat	Incoming messages should not be received and printed on the chat	Incoming messages are not received and printed on the chat	Pass

TC07	Leaving a chat	When the user leaves a chat	The user should send a request and receive a confirmation to enter the chat again	The user sends a request and receives a confirmation to enter the chat again	Pass
TC08	Chats not sent	When a chat is not received	The user should be notified when the message sent is not received	The user is not notified when the message sent is not received	Fail

Below is an illustration of this project where two mobile phones are exchanging messages using the mobile application developed for the network system created.

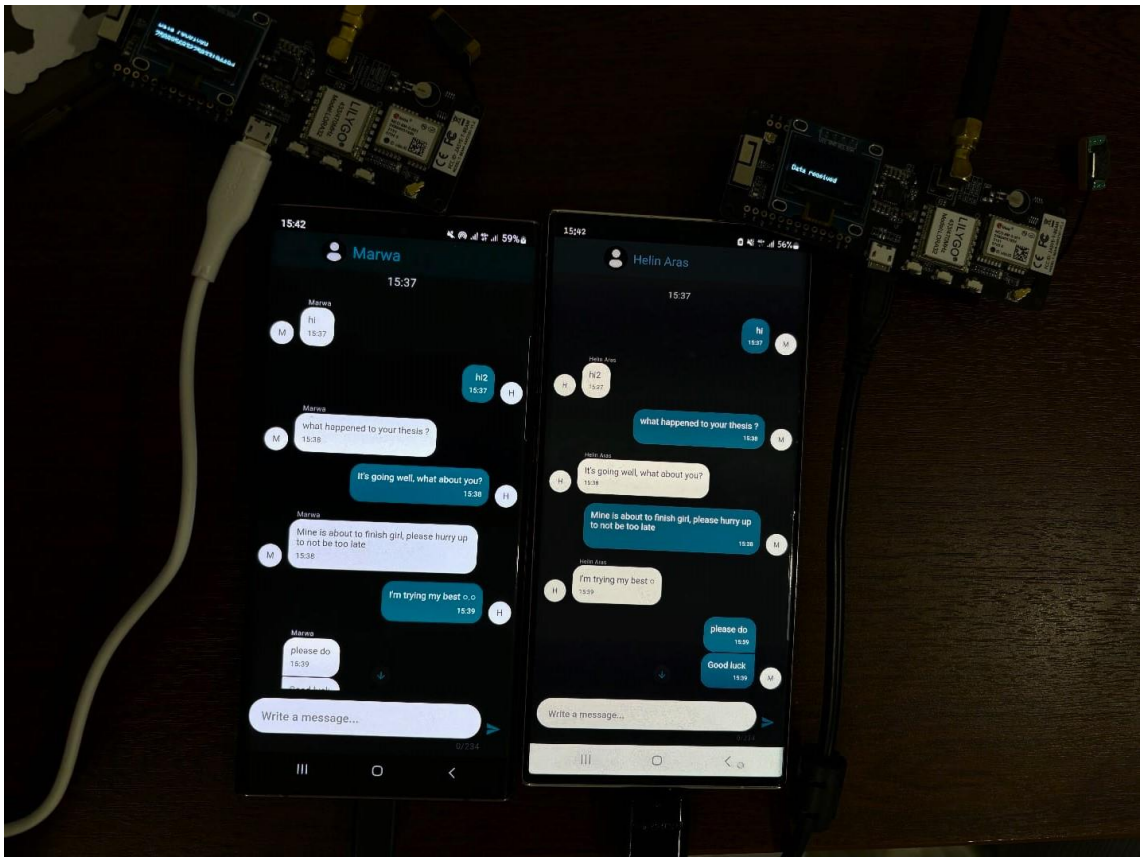


Figure 5.5 The developed mobile application

Chapter 6 Conclusion and Future Works

6.1 Conclusion

This project broadens the spectrum of networking and its usage in the field of data communication. Meanwhile, the data communication scheme is becoming more and more centralized, with cloud computing, and other forms of internet-connected servers where data is stored on core servers that can be accessed using an integrated network like the internet. However, despite the internet being an international network, there are blind spots where it does not reach, or the signal is weak and inefficient. This is especially the case in rural areas. Therefore, this viewpoint has motivated the birth of this project. The main purpose of this project, the Implementation of a portable network through using small transceivers to transmit data, is to construct a network that provides data communication that can be easily carried and utilized anywhere. The reason behind choosing the LoRa modulation transceiver as the medium between users in the system was mainly because of the long-range connectivity, low power consumption, and its lightweight device which can be carried anywhere. Ultimately, the idea of this project is successfully turned into a product that can successfully transmit data over a stable and secure network. Like any other engineering product, the work done on this project is just the basis of this idea and therefore can be further optimized by adding additional features.

6.2 Future Works

The implementation of a portable network using LoRa is a highly modular project that integrates multiple fields of technology. Hence, the scope of maintenance and improvement of the outcome product of this project is broad and flexible. Below are some of the add-ons and future works that can be incorporated with the product of this project:

- GPS tracking: The TTGO T-Beam board used for this project, is integrated with a GPS chip. This chip can be used to track the user’s location and display it on the mobile application. This feature is exceptionally useful to detect the user's location in case of no response from the user.

- Group chats: The application created for this project, only has user-to-user chats. Therefore, a group chat feature is a reasonable add-on to the application. This is particularly useful when a group of people are using the application and therefore need to communicate as a group.

- Data packet additional features: the data packet header designed for this project has identifiers of sender and receiver and the code byte. Additionally, other features can be added as well like acknowledgment (ACK), session timeout, and error detection (checksum).

References

Bembe, M., Abu-Mahfouz, A., Masonta, M. and Ngqondi, T. (2019). "A survey on low-power wide area networks for IoT applications." Telecommunication Systems, 71(2), pp.249–274.

Bouras, C., Gkamas, A., Kokkinos, V. and Papachristos, N., 2019, October. Using LoRa technology for IoT monitoring systems. In *2019 10th International Conference on Networks of the Future (NoF)* (pp. 134-137). IEEE.

Cardenas, A.M., Nakamura Pinto, M.K., Pietrosemoli, E., Zennaro, M., Rainone, M. and Manzoni, P., 2020. A low-cost and low-power messaging system based on the LoRa wireless technology. *Mobile networks and applications*, *25*, pp.961-968.

Centenaro, M., Vangelista, L., Zanella, A. and Zorzi, M., 2016. Long-range communications in unlicensed bands: The rising stars in the IoT and smart city scenarios. *IEEE Wireless Communications*, *23*(5), pp.60-67.

Chaudhari, B.S. and Zennaro, M. eds., 2020. LPWAN Technologies for IoT and M2M Applications. Academic Press.

De Almeida, I.B.F., Chafii, M., Nimr, A. and Fettweis, G., 2020, December. In-phase and quadrature chirp spread spectrum for IoT communications. In GLOBECOM 2020-2020 IEEE Global Communications Conference (pp. 1-6). IEEE.

El Alami, M., Benamar, N., Younis, M. and Shahin, A.A., 2017, June. A framework for hotspot support using Wi-Fi direct based device-to-device links. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)* (pp. 552-557). IEEE.

Falanji, R., Heusse, M. and Duda, A., 2022, November. Range and Capacity of LoRa 2.4 GHz. In International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services (pp. 403-421). Cham: Springer Nature Switzerland.

Gambi, E., Montanini, L., Pigini, D., Ciattaglia, G. and Spinsante, S., 2018. A home automation architecture based on LoRa technology and Message Queue Telemetry Transfer protocol. *International Journal of Distributed Sensor Networks*, *14*(10), p.1550147718806837.

Green, P.E. (1982). *Computer Network Architectures and Protocols*. Springer eBooks.

Höchst, J., Baumgärtner, L., Kuntke, F., Penning, A., Sterz, A. and Freisleben, B., 2020, May. Lora-based device-to-device smartphone communication for crisis scenarios. In *Proceedings of the 17th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*.

Kulkarni, P., Hakim, Q.O.A. and Lakas, A., 2019. Experimental evaluation of a campus-deployed IoT network using LoRa. *IEEE Sensors Journal*, *20*(5), pp.2803-2811.

Leiner, B.M., Cerf, V.G., Clark, D.D., Kahn, R.E., Kleinrock, L., Lynch, D.C., Postel, J., Roberts, L.G. and Wolff, S.S. (1997). "The past and future history of the Internet." Communications of the ACM, 40(2), pp.102–108.

Li, F., Wang, X., Wang, Z., Cao, J., Liu, X., Bi, Y., Li, W. and Wang, Y., 2020. A local communication system over Wi-Fi direct: Implementation and performance evaluation. *IEEE Internet of Things Journal*, *7*(6), pp.5140-5158.

Mekki, K., Bajic, E., Chaxel, F. and Meyer, F., 2019. A comparative study of LPWAN technologies for large-scale IoT deployment. ICT express, 5(1), pp.1-7.

Mikhaylov, K., Stusek, M., Masek, P., Fujdiak, R., Mozny, R., Andreev, S. and Hosek, J., 2020, June. Communication performance of a real-life wide-area low-power network based on Sigfox technology. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)* (pp. 1-6). IEEE.

Prakosa, S.W., Faisal, M., Adhitya, Y., Leu, J.S., Köppen, M. and Avian, C., 2021. Design and implementation of LoRa based IoT scheme for Indonesian rural ar-

ea. *Electronics*, *10*(1), p.77.

Raza, U., Kulkarni, P. and Sooriyabandara, M., 2017. Low power wide area networks: An overview, ieee communications surveys & tutorials 19 (2): 855–873.

Sarwar, A., Li, B. and Dempster, A., 2009, December. SPOT in location based emergency services, LBES detailed analysis. In *International Global Navigation Satellite Systems Society Symposium (IGNSS), Qld, Australia* (pp. 1-15).

Sciullo, L., Fossemo, F., Trotta, A. and Di Felice, M., 2018, December. Locate: A lora-based mobile emergency management system. In *2018 IEEE global communications conference (GLOBECOM)* (pp. 1-7). IEEE.

Tsavalos, N. and Abu Hashem, A., 2018. Low power wide area network (LPWAN) Technologies for Industrial IoT applications.

Vangelista, L., 2017. Frequency shift chirp modulation: The LoRa modulation. IEEE signal processing letters, 24(12), pp.1818-1821.

Wikipedia. (2019). Internet of things. [online] Available at: https://en.wikipedia.org/wiki/Internet_of_Things.

Yefremov, Y. TTGO T-Beam V1.1. Available at: https://doc.riot-os.org/group__boards__esp32__ttgo-t-beam.html#details.

Abstract - Arabic خلاصة

لا يزال توافر تغطية الشبكة هدفًا لم يتحقق في العديد من المواقع. يواجه الأشخاص الذين يعيشون أو يعملون أو حتى يتجولون في مناطق نائية بدون بنية تحتية للشبكة صعوبة في التواصل حتى يومنا هذا. نظرًا لأن نشر أبراج الشبكة والبوابات قد يكون مكلفًا للغاية، فقد اكتسبت حلول أخرى مثل الشبكات الخاصة أو الشبكات التي لا تحتوي على بنية تحتية والتي تعتمد على الاتصال بين العقد التي لا تحتوي على بوابات شعبية كبيرة. ظهرت شبكات المنطقة الواسعة LPWAN كواحدة من التقنيات الرائدة في صنع مثل هذه الأنظمة. تجعل خصائص (LPWANs) منخفضة الطاقة مثل استهلاك الطاقة المنخفض والاتصال بعيد المدى والصفات الأخرى منها الخيار المثالي للأنظمة المقترحة. يحل يبني المشروع تطبيقًا .LPWAN LoRa هذا المشروع هذه المشكلة من خلال تطوير نظام مراسلة يعتمد على تقنية متصلة بالجهاز ونقلها إلى المستقبل LoRa محمولًا يسمح للمستخدمين بإرسال رسائل نصية يتم إرسالها إلى وحدة على الأمان وبروتوكولات الاتصال بالبيانات الضرورية، وبالتالي، ستصبح مصدرًا LoRa المقصود. ستعتمد وحدة موثوقًا للاتصالات عندما تكون الشبكات الأخرى غير متاحة.

Abstract - Kurdish پوختە

بەردەستبوونی ڕووماڵکردنی تۆڕەکان هێشتا ئامانجێکە کە لە زۆر شوێندا بەدەست ناهێنرێت. ئەو کەسانەی کە لە ناوچە دوورەکاندا دەژین، کار دەکەن، یان تەنانەت گەشت دەکەن کە ژێرخانی تۆڕیان نییە، تا ئەمڕۆش کێشەیان هەیە بۆ پەیوەندیکردن. بەو پێیەی کە جێگیرکردنی تاوەرەکانی تۆر و دەروازەکان ڕەنگە زۆر گران بێت، چارەسەرەکانی دیکەی وەک تۆڕە تایبەتەکان یان تۆڕەکانی بێ ژێرخانی کە لەسەر بنەمای پەیوەندی نێوان گرێکانن وەک (LPWAN) کە دەروازەیان نییە، خەریکە ناوبانگیان بەدەستهێناوە. تۆڕەکانی ناوچەی فراوانی کەم کارەبا یەکێک لە تەکنەلۆژیا پێشەنگەکان لە دروستکردنی ئەم جۆرە سیستەمەدا سەرهەڵدەدەن. تایبەتمەندییەکانی وەکو بەکارهێنانی کارەبای کەم، پەیوەندی مەودای دوور و کوالیتیەکانی تر وای لێدەکات ببێتە LPWAN هەڵبژاردەیەکی ئایدیاڵ بۆ سیستەمە پێشنیار کراوەکان. ئەم پرۆژەیە ئەم کێشەیە چارەسەر دەکات بە پەرەپێدانی پرۆژەکە بەرنامەیەکی مۆبایل دروست .LPWAN LoRa سیستەمێکی پەیام ناردن لەسەر بنەمای تەکنەلۆژیای کە بەستراوەتەوە LoRa دەکات کە ڕێگە بە بەکارهێنەران دەدات نامەی کورت بنێرن کە دەنێردرێت بۆ مۆدیوڵێکی لەسەر بنەمای ئاسایش و پرۆتۆکۆڵە LoRa بە ئامێرەکەوە و دەگوازرێتەوە بۆ وەرگرەکەی مەبەست. مۆدیۆڵی پێویستەکانی پەیوەندی داتا دەبێت و هەربۆیە، دەبێتە سەرچاوەیەکی متمانەپێکراو بۆ پەیوەندیکردن کاتێک تۆڕەکانی تر بەردەست نین.