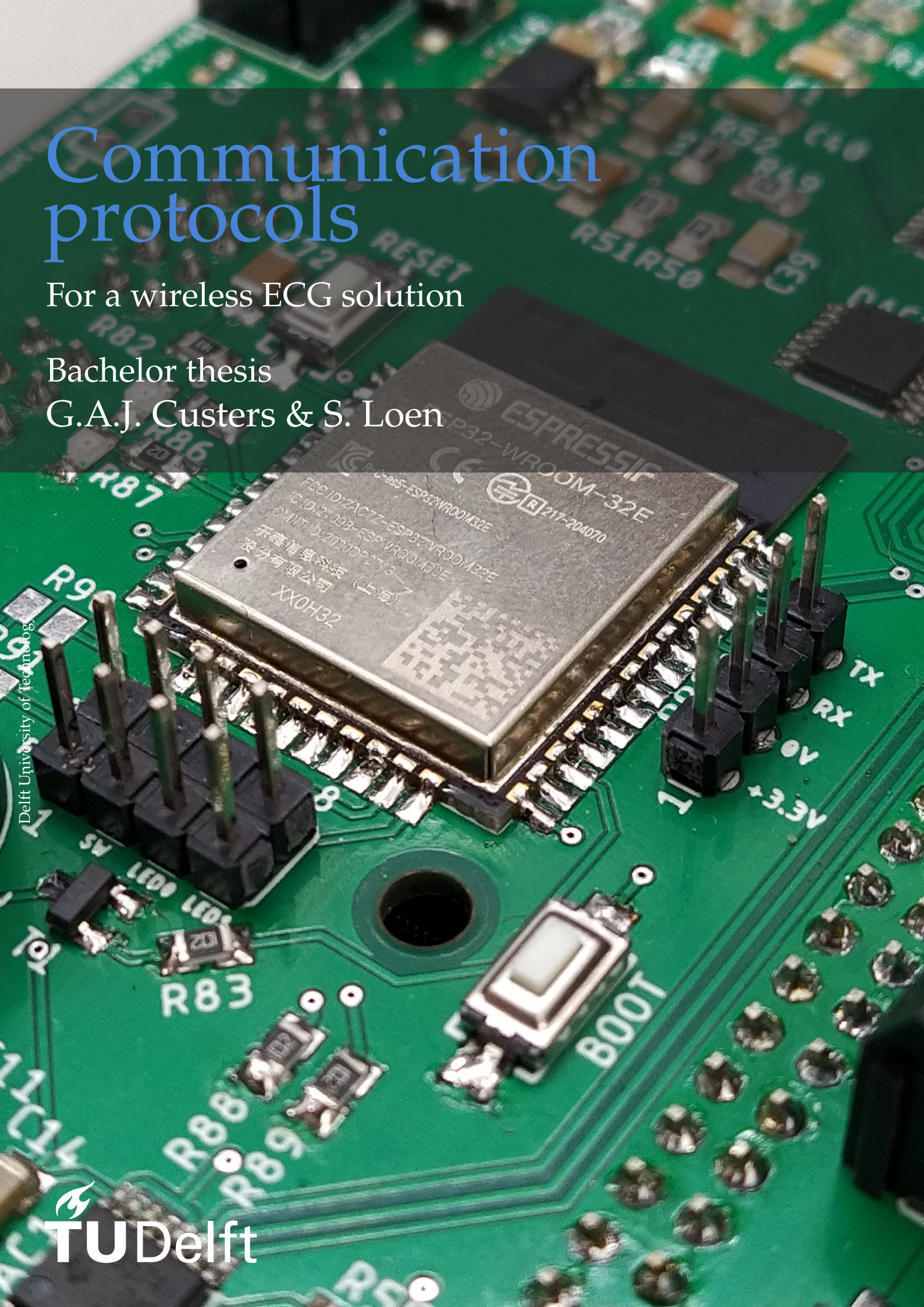
Communication protocols

For a wireless ECG solution

Bachelor thesis

G.A.J. Custers & S. Loen



Delft University of Technology

TU Delft

Communication protocols

For a wireless ECG solution

by

G.A.J. Custers & S. Loen

Student Name	Student Number
Geert Anne Jan Custers	5119995
Stefan Loen	5115973

Instructor:	Prof. Dr. Ir. W. Serdijn
Project Duration:	April, 2022 - June, 2022
Faculty:	EEMCS, Delft

TU Delft

Abstract

This report is part of the WiECG project. The goal of the WiECG project is to create a prototype device that makes it possible to perform a 12-lead ECG on patients without wires from a patient to the monitor. The solution consists of a transmitter and receiver of which one is close or on the patients body and the other is connected to a monitor.

This thesis describes the design and implementation process for the communication protocols. This concerns mainly the WiFi communication from transmitter to receiver and the I2C pairing protocol. The WiFi communication protocol was extended with retransmission and encryption to ensure the reliability and discreteness of the data, which was indicated to be of high priority by ambulance personnel. The channel needs to minimally support 9 samples of 16 bits with a frequency of 500 samples per second. The pairing protocol needs to enable the transmitter and receiver to pair in mere seconds.

A prototype is made which successfully pairs and transfers data from transmitter to receiver. This prototype still needs to be tested thoroughly.

Preface

This thesis was completed in cooperation with two other groups. Roy van Krieken and Pelle Wiersma took care of the hardware part. Keyvan Khalili and Sebastian Speekenbrink covered the digital signal processing. Sebastian came up with this project, brought us together and succesfully convinced us to tackle this problem.

We greatly appreciated the motivation of every single team member. Even though the final report is split in 3 parts, we really did this together as a team. It sometimes made it hard to tell who's responsibility a certain task is, but after a while everyone just naturally picked up certain tasks. As the group worked together in almost all cases, documenting the work in a logical flow was quite the challenge. We therefore recommend to read the theses in the following order: Protocol, Digital Signal Processing [1] and Hardware [2].

Special thanks go to Wouter Serdijn who supervised but mostly supported us in our process. We thank Leo Roos, Jim van Akkeren, Paul Broers, Jaimie Dik and Hans Schuitmaker for providing us valuable information about the world of ambulances and ECG, and for delivering us some necessary materials for the prototype. We appreciated the feedback of Asli Boru and Francesc Varkevisser and furthermore thank Martin Schumacher and Ton Slats for facilitating and aiding in the assembly.

G.A.J. Custers & S. Loen
Delft, June 2022

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
ACK	Acknowledge
API	Application Programming Interface
BER	Bit error rate
BSS	Basic service set
CRC	Cyclic redundancy check
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
DHCP	Dynamic Host Configuration Protocol
DSP	Digital Software Processing
DSSS	Direct Sequence Spread Spectrum
ECG	Electrocardiogram
ESP-IDF	Espressif IoT Development Framework
ESP32	The ESP32 microprocessor from Espressif
FHSS	Frequency Hopping Spread Spectrum
FIFO	First In First Out
FreeRTOS	Free Real-time Operating System
I2C	Inter-integrated circuit
IPv4	Internet Protocol version 4
LLVM	Low level virtual machine
LMK	Local Master Key
MAC	Media Access Control, address of a NIC.
NIC	Network-Interface Controller
OOB	Out of Band
O-QPSK	Offset Quadrature Phase Shift Keying
PMK	Primary Master Key
RAM	Random Access Memory
SCL	Clock signal of I2C connection
SDA	Data signal of I2C connection
SDK	Software Development Kit
SSID	Service Set Identifier
WLAN	Wireless Local Access Network
WiECG	Name for Wireless ElectroCardioGram project
WiFi	Name for international standard IEEE 802.11

1 Introduction

This thesis was written in collaboration with 6 people. The project was divided in three subgroups, each delivering their own thesis. For this reason some parts in this thesis will mimic those of other groups (e.g. General Introduction, problem statement, proposed solution and state of the art). To preserve the logical flow of the design, the order of reading should be as follows: Protocols, Digital Signal Processing [1] and Hardware [2].

1.1. The WiECG Project

In the Netherlands alone, 1.3 million ambulance rides are made each year . Of these 1.3 million rides 76% are urgent [3]. After interviewing Jim van Akkeren (Operational Head Witte Kruis Ambulance Zorg Den Haag) and Mirthe Ruijgrok (Ambulance operator) it was concluded that in 90% of the cases an ECG is connected to the patient being transported. An ECG is used in many cases to exclude a heart related problem as the treatment of such should happen as fast as possible. Furthermore ECGs are used when any form of anaesthesia or medicine is administered, to monitor the patients reaction.

As the deployment of an ECG requires wires, problems arise for the ambulance personnel in applying them. The Wireless ElectroCardioGram project aims to replace these wires with a wireless solution.

1.1.1. Basics of ECG

An ECG is a visualization of the muscle contractions produced by a heart. This is done by measuring the vector projection of the hearts' electrical field on the chest of a patient. The measurement is done using electrodes located on the body of the patient, whose potentials are caused by these contractions. These potentials produced by the heart are then registered and visualized on a monitor.
There are two main ECG variations health workers employ [4]. One with 4 electrodes and one with 6 additional ones. The first configuration is called the extremity electrodes, which can give a general electrical overview of the heart functions. In the second configuration 6 other electrodes are added: the chest electrodes. These give more detailed information of the heart on which diagnoses can be made [5]. With these signals 12 signatures in total can be obtained.

1.1.2. Problem statement

As shown before, the usage of ECGs by ambulance personnel is crucial to ensure the well being of the patients. ECGs are currently applied by usage of electrodes attached to the chest of the patient. For regular, non emergency use, only the extremity electrodes are used. In emergency situations a full 12 lead (10 electrodes/wires) configuration is used. Currently these electrodes consist of stickers connected to wires which are connected to a heart monitor. According to the interviewed ambulance personnel, the wires are very annoying to work with in emergency situations. The wires get tangled, dirty and in the way of the ambulance personnel as it obstructs the cabin.

1.1.3. Proposed solution

The proposed solution is a device where the same monitor can be used as before, but where the cables have been replaced by a pair of wireless devices. The transmitter device has 10 electrodes which have to be applied to the patient like before, requiring no extra actions for the operator. The receiver side can be plugged into the monitor, also requiring no extra actions. This plug n' play system can be used with any ECG monitor as long as the connector for that monitor is available. To make sure that the devices transmit and receive the right signal and not that of another pair, they can be paired easily by having the transmitter and receiver briefly connect with each other. This solution solves the problems mentioned in Section 1.1.2 in the following way:

- The short cables tend to tangle up way less compared to the longer ones.
- Because the transmitter device is located near the patient and the receiver lies next to the ECG monitor, no cables are suspended through the ambulance, greatly improving the comfort/workflow of the operator.

To achieve this goal, a self proposed electrical engineering bachelor graduation project was submitted to Delft University of Technology. This project is executed with 6 others, and is split up into three parts (as can also be seen in Figure 1.1):

- Protocol (PROT)
  The group that deals with the wireless transmission of the ECG data
- Digital Signal Processing (DSP)
  The group that process the signal digitally and forwards it to the wireless module
- Hardware (HW)
  The group that prepares the measured signals for digital conversion and facilitates the aforementioned groups in creating a prototype device.

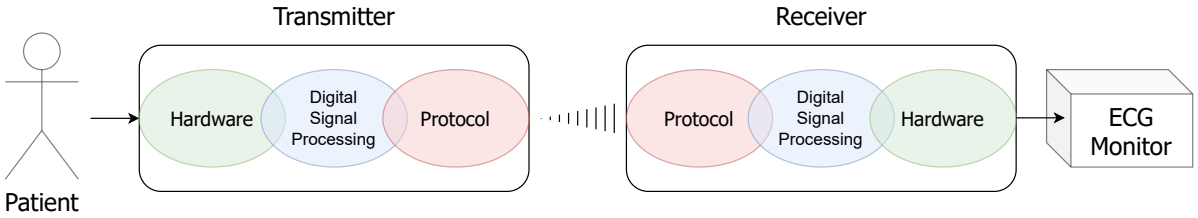


Figure 1.1: Brief overview of the solution and the separation of tasks

1.1.4. State of the art

Currently wireless heart monitoring (via ECG)is mainly used in several markets; medical and consumer. Consumer wireless ECG systems (e.g. KardiaMobile 6L™[6]) are usually limited to 6 leads, or only one in case of wearable devices (e.g. *Apple Watch™, Galaxy Watch™*), which output the data to a smartphone or display. Furthermore, the sensors and workflow used to obtain the ECG differ majorly from those medical personnel uses. (No stickers but handheld/wearable sensor device).

The medical market is home to devices which are purpose-made for health monitoring on a diagnostic or treatment level. Currently, the Lifepak 15™[7] is in use in many ambulances to monitor the vitals of the patient. This device is part of a production line which is 12 years old and is still in use - in Dutch ambulances - due to the others[1] not being reliable (slow to start up, easily breakable or simply not working.) The connections to the monitor of these currently used monitors are however wired, leading to the aforementioned problems.

However, there are medical wireless monitoring systems on the market that remove the wires between patient and monitor. Some examples are the ZOLL Heart Failure Management System (HFMS)™[9] and the Corpuls3™[10]. However, the HFMS focusses on detection of heartattacks in a non emergency setting

[1]Jim van Akkeren stated that since 2017 the Philips Tempus ALS™[8] monitor was in use. But most ambulance regions stopped usage due to reliability complaints

and the Corpuls3 simply has a detachable wireless display (the relatively bulky defibrilator/patient box of around 3 kg has to be close to the patient). None of the devices researched focused on altering the medium of the signal like the proposed solution.

1.2. The scope of this subgroup

The solution mentioned in the previous section requires stable and reliable communication between transmitter and receiver. The protocol subgroup deals with the wireless transmission. Besides the wireless communication protocol, this also includes the method of pairing the receiver and transmitter.

The communication protocol subgroup is responsible for the following tasks

- Transferring data from transmitter to receiver without data loss and without inducing a high latency.
- The transmitter and receiver need to be paired with each other to prevent unwanted connection to an unknown and potentially malicious device.
- Handling the data securely. The data needs to be encrypted to prevent eavesdropping.

2

Program of requirements

Here the most important requirements are listed. Throughout the report, these requirements will get referenced to ensure that the design choices that were made have value for the end result of this project. The design described in this report and the reports of the other subgroups will attempt to completely fulfill the following list:

- **G1:** The device must not employ wires from patient to monitor.
- **G2:** The device must allow the user to perform a 12 lead ECG.
- **G3:** The device must be safe for the patient
- **G4:** The data handled by the device must be safeguarded
- **G5:** The device must not induce a delay of more than 5 seconds to the workflow of the user compared to a regular ECG wire.
- **G6:** The signal transferred by the device must be indistinguishable by the eye from the signal transferred by a regular ECG wire.
- **G7:** The device must have a battery life of 2 hours.
- **G8:** The device must not be bigger than 10cm x 20cm x 5cm (a smartphone-device)
- **G9:** The device must be lighter than 500 gram
- **G10:** A prototype device must be functional within 10 weeks from the start of the project.
- **G11:** The prototype must not cost more than 500 euros.

From this list, the following requirements were specified which specifically apply to the protocol part of this project.

- **G1P:** The device uses a digital wireless communication protocol.
- **G2P:** The communication protocol must handle 9 concurrent signals.
- **G4P.a:** The communication channel must be protected from eavesdropping.
- **G4P.b:** The device must ensure safe connection between transmitter and receiver
- **G4P.c:** The pairing must be easy to confirm.
- **G5P.a:** The pairing of the device must work within 2 seconds.
- **G5P.b:** The transmitter and receiver must be interchangeable.
- **G6P.a:** The device must transfer all signals lossless.
- **G6P.b:** The communication must not induce a delay of more than 0.5 seconds.
- **G10P:** The communication module must be available within the timespan.

On top of this list, some requirements come from the other subgroups, but apply to the protocol part. These requirements are listed here:

- **G6D.a:** An input signal must be sampled at 500 Hz
- **G6D.b:** An input signal must be sampled at 16 bits resolution
- **G10H.a:** The utilized microprocessor must be hand-solderable.

2.1. Reliability

One of the most important design goals of the final product is reliability. State of the art ECG monitor designs have seen minimal adoption because they aren't reliable enough. I.e., when the device is powered on, there is uncertainty if the device will function correctly. This problem is further substantiated in the introduction of the report.

Therefore, to differentiate our final product from other existing solutions, throughout the project there is an emphasis on reliability. The product must always work, and any failing conditions must be analysed and handled appropriately. However, this requirement cannot be reasonably tested within the 10 week time frame that is allocated to the BAP. Hence, it is not included in the list of requirements. Regardless, reliability is an important design principle throughout the project.

3

System overview

The proposed solution in Section 1.1.3 can, in accordance to the program of requirements, be further specified as shown in Figure 3.1. This also depicts a high-level overview of the contributions of each subgroup. This report relates to the protocol part. Refer to the report of DSP [1] and hardware[2] for a broader understanding of the WiECG project. In Section 3.1 a detailed overview is given of the dataflow from transmitter to receiver, which is of interest for the protocol subgroup. Finally, in Section 3.2, the complete system is shown where the detailed overviews are connected to the detailed overviews of other subgroups.

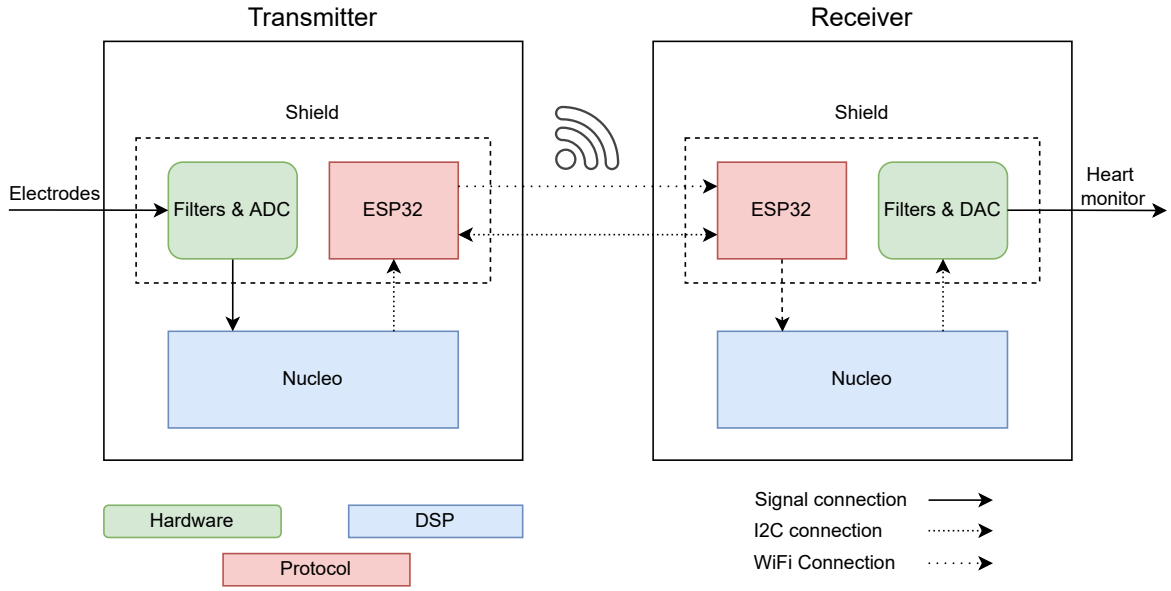


Figure 3.1: Brief overview of the solution and the separation of tasks

3.1. Dataflow overview

Figure 3.2 provides a complete overview of the different modules on the ESP32. The inputs of this subsystem are an I2C connection with ECG data and a button on the transmitter side. The outputs are a buzzer and I2C connection with ECG data on the receiver side.

The design of ECG data transmission and receival is discussed in Section 4.1. More about the implementation of the queue, bitmap, buffer and timer can be found in section 5.2.3, 5.4.1, 5.4.2 and 5.5 respectively.

The design of the pairing mechanism is discussed in Section 4.2 and the implementation is discussed in Section 5.2.2.

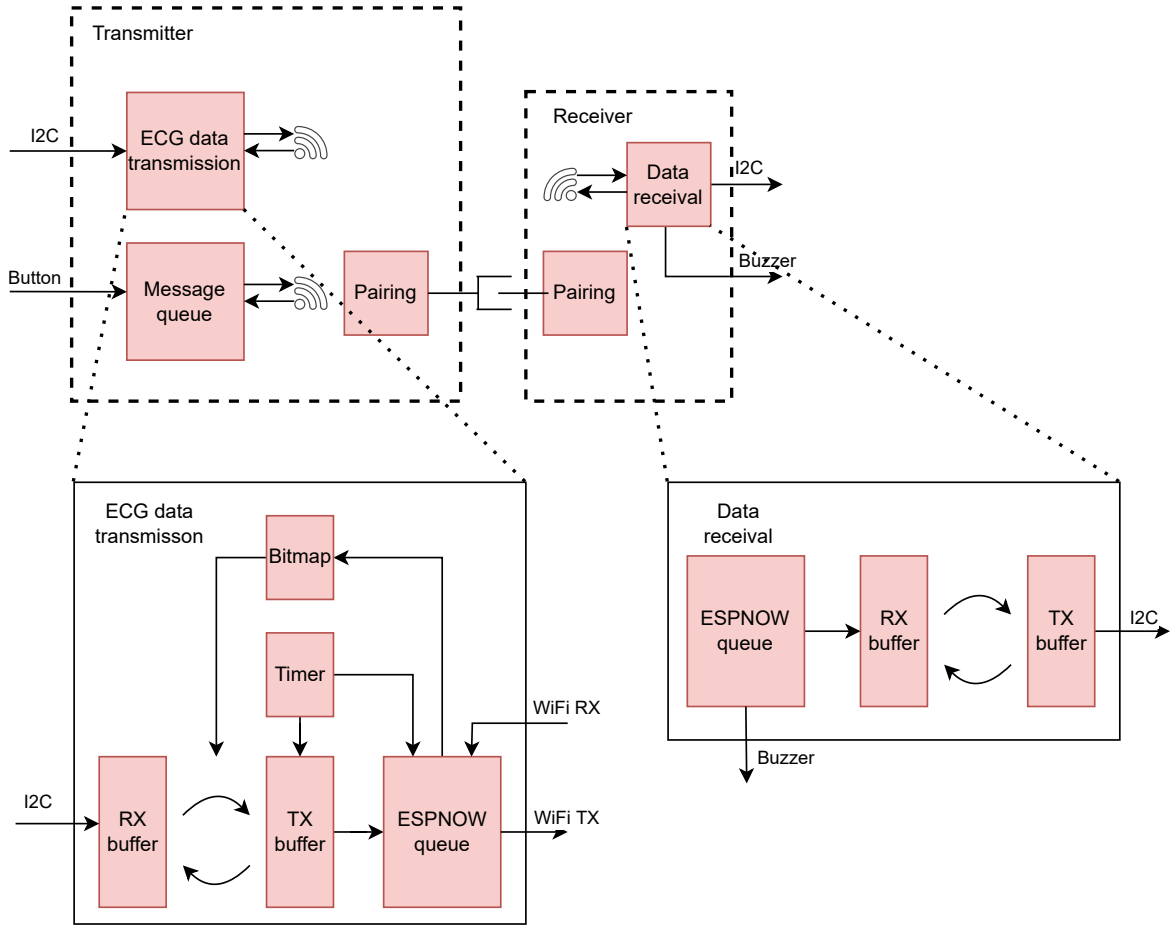


Figure 3.2: Overview dataflow from transmitter to receiver

3.2. Complete overview

Figure 3.3 shows the protocol subgroup's responsibility placed in the complete overview of the system. For complete understanding of this figure, refer to the report of DSP[1] and Hardware[2]

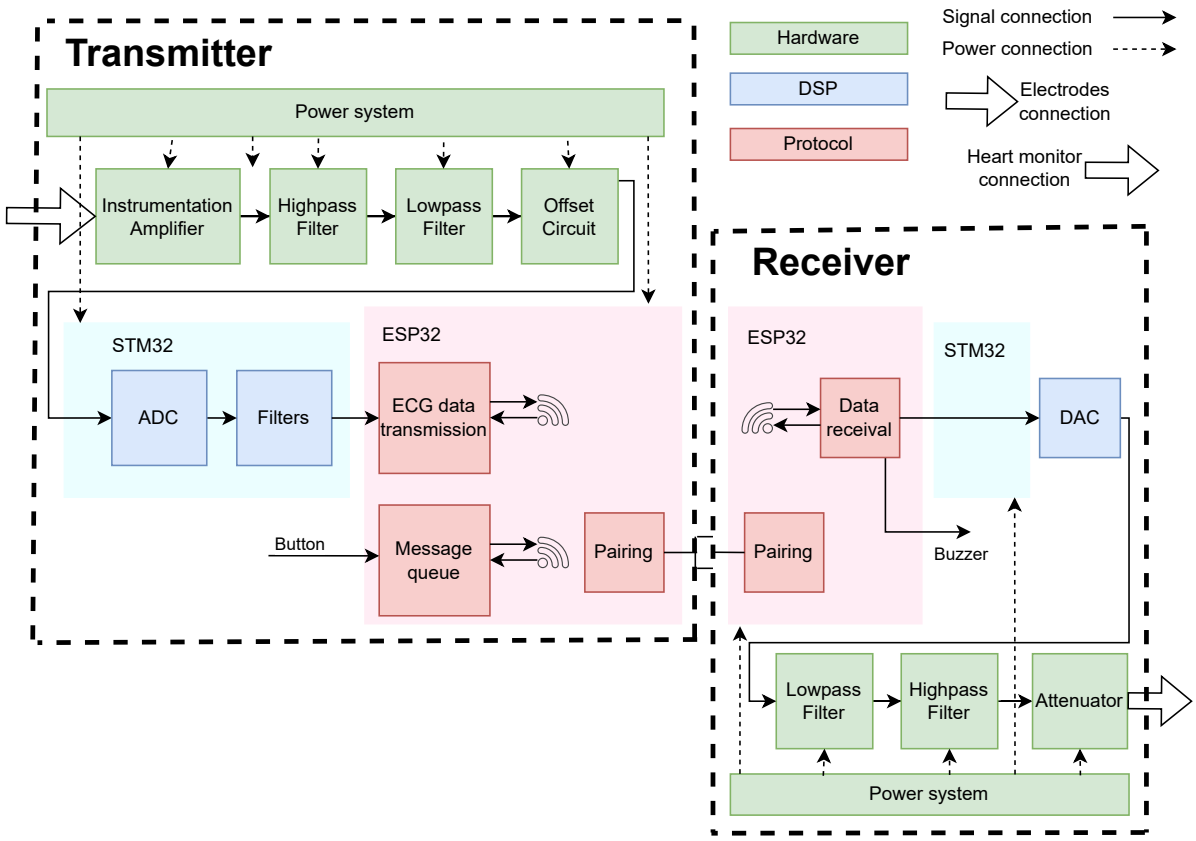


Figure 3.3: Complete overview of the WiECG project

4

Design

4.1. Transmission protocol

This section discusses several transmission protocols that comply with the most important requirement G1P (The device uses a digital wireless communication protocol.). These protocols were all taken into consideration when choosing a protocol for the device. Of every protocol, the specifications, pros and cons are stated. A short summary is found in the following list.

- **UWB** - Ultra Wide Bandwidth transmits data using very short pulses and therefore over a very large bandwidth. An advantage is low power density (in the order of μW), and high data rate. UWB is mainly used in high data rate applications.
- **WiFi** - IEEE 802.11b (WiFi) operates in the 2.4Ghz band, supporting a high data rate. Latency is relatively low, making it suitable for carrying multi-lead ECG real-time ECG signals. A drawback is high transmission power, of 10-100 mW.
- **Bluetooth** - Bluetooth operates in the 2.4Ghz band, supports a generally low transmission range and low data throughput of less than 1 Mbps. Power usage depends on transmission range but can vary from 1-100 mW. One drawback is that other 2.4GHz emitters might compete for transmission time and significantly increase latency, and reduce throughput.
- **Zigbee** - Zigbee also operates in the 2.4Ghz band , supporting a low data rate of < 250 Kbps, at a low range of 1-10m. It has low power requirements of <1 mW, therefore being suited to applications with relaxed latency requirements.

4.1.1. UWB

UWB is a transmission protocol that uses very short pulses for transmission, occupying a very large bandwidth (typically 500 MHz) [11]. It operates in the 3.1 - 10.6 GHz spectrum, meaning it intrudes on other wireless communication technologies. Hence, it's transmission power is very low, namely at -41.6 dBm/MHz, which is the level at which other wireless technologies classify a signal as noise. Because of this low transmission power, UWB is best suited for short range applications. O. Flink [11] describes that various modulation and multiple access schemes can be used with UWB.

4.1.2. WiFi

WiFi is an umbrella term for a set of standards defined by the IEEE under IEEE802.11. The standards define a wide variety of applications of wireless communication, at different transmission speeds. Hence it is difficult to provide a comprehensive overview of the entire technology. WiFi acts in multiple frequency bands, but focus is on the 2.4 GHz band since the hardware is more common. In the 2.4 GHz band there are 14 communication channels defined, each having a separation of 5 MHz. To divide access to each channel, CSMA/CA is used. With CSMA/CA WiFi stations first assess whether the channel is available. If the channel is busy, the station waits, and then waits for a random back-off interval, to avoid a race on transmission when the channel is available again.

To harness itself from interference, WiFi makes use of DSSS. In DSSS, a binary code known by both the transmitter and receiver is overlayed on top of the modulated signal, to increase the signal's bandwidth. When the receiver de-spreads this signal using the same binary code, the original signal will be intact, while interference will be suppressed. As concluded by E. McCune [12], DSSS is successful at reducing in-band interference. A property that is very beneficial, since interference in the 2.4 GHz band is expected.

Traditional WiFi
Traditionally, the communication protocol of WiFi utilises BSSs [13]. This provides the service of a WLAN, meaning that there is a principle of locality (i.e. some devices are and some devices are not members of the network). This scheme is usually supported by a Network Layer protocol such as IPv4.

When utilising WiFi this way, devices cannot simply start sending to each other, they first need to perform an association procedure. This procedure requires sending an "Association Request" frame, and receiving an "Association Response" frame. This requires several round trips, before data can be sent. After association, IP addresses must be allocated, which is usually done with a DHCP server. This adds considerable overhead, while the added functionality of WLANs are not required for the simple case of transmitting data from one device to another.

ESP-NOW
ESP-NOW [14] is a connectionless communication protocol implemented on top of WiFi. The protocol implements communication on the Data Link Layer, using WiFi as a carrier. The main advantage of the protocol is that it is connectionless, which means that there is no context in which a packet is sent. An ESP-NOW packet is addressed to a particular MAC address, meaning there is no concept of a "network" that a device needs to connect to. This vastly reduces the amount of complexity required for communicating, as there is very little overhead involved. Since this communication protocol also does not require a Network Layer implementation, there is no need to run a DHCP server, or to allocate IP addresses.

ESP-NOW also natively supports encryption. The encryption scheme uses two keys, the PMK (Primary Master Key) and LMK (Local Master Key). The PMK is used to encrypt the LMK with the AES-128 encryption algorithm. The LMK is then used in the CCMP encryption scheme. CCMP is a block cipher, data is encrypted in fixed-size blocks. Each block depends on the previous block, creating an interdependence between encrypted blocks. Devices need to agree on both the PMK and LMK to successfully transmit encrypted data to each other. The inclusion of encryption in the protocol satisfies requirements **G4P.a** and **G4P.b**, which are both about safe and protected transmission of the ECG data.

Additionally, ESP-NOW peering can easily be performed out of band. Quick, easy and reliable pairing falls under requirement **G5P.a**. To pair two devices, the devices need only to exchange their MAC addresses, PMK and LMK. After this, the two devices are paired, and can immediately transfer data. This method of pairing is simple, and does not include many round trips or handshake procedures, meaning there are limited error conditions, fitting in a reliability-conscious design.

4.1.3. Bluetooth
Bluetooth is a popular communication protocol used for embedded devices. It operates in the 2.4GHz band, with 79 bluetooth channels allocated. Each channel is separated by 1 MHz. As opposed to WiFi, bluetooth uses FHSS to accomplish interference hardness. Where DSSS is interference suppression, FHSS acts more as interference avoidance. FHSS spreads the modulated signal by hopping through a pre-selected sequence of channels, at different frequencies. E.g. a packet can be sent on channel 6, and the next one can be sent on channel 18. This way the receiver can apply a narrow bandpass filter on the channel, blocking interference. However, if a channel with interference is chosen for transmission, the interference falls within the passband of the bandpass filter. Hence, this interference avoidance scheme works well for out of band interference, but can be sensitive to in band interference. [12].

4.1.4. ZigBee

ZigBee is a very low power, low data-rate transmission protocol. [15] It is intended for similar application as bluetooth, but where the data rate requirement is lower, and the expected battery of the device must be high. The protocol operates in the unlicensed 2.4 GHz bandwidth. When operating at 2.4 GHz, ZigBee uses the O-QPSK binary modulation technique. To perform multiple access, ZigBee uses CSMA-CA.

4.1.5. Making a choice

Technology	OOB pairing	Power	Range	Data Rate	Interference	Error Detection
IEEE 802.11b	Yes	10 - 100 mW	100 m	High (100 Mbps)	Low	CRC
Bluetooth LE	Yes	Low (1 - 10 mW)	< 100 m	up to 1 Mbps	Relatively High	CRC
ZigBee	Yes	Low (< 1 mW)	1-10 m	250 Kbps	Relatively High	CRC
UWB	Yes	Very Low	10 m	Up to 200 Mbps	Relatively High	Not by itself

Table 4.1: Technology Analysis

Using Table 4.1 and context surrounding the technologies, a decision can be made on which wireless protocol is used. Firstly, UWB is not a good choice for the given application. The application requires high reliability (i.e. consistent latency and adequate data rate), and not necessarily high data rate. These are conditions that UWB does not meet.

ZigBee is an interesting protocol because of its lower energy usage. However, out of all the protocols its data rate is the lowest. Additionally, research has shown that the bit error rate of ZigBee increases drastically when under interference from other Bluetooth and WiFi devices [16], a situation which is an expected use-case. While the advertised bandwidth is adequate under ideal conditions, there seems to be no guarantee that these ideal conditions will be reached. Under non-ideal conditions ZigBee can lose almost half of its performance [17], which is sufficient for transmission of the ECG signal, but such large drops in performance make ZigBee difficult to use in reliability critical situations.

Another lower energy usage protocol is Bluetooth LE (Low Energy). While having slightly higher power requirements than ZigBee, it also has a larger data rate. However, while the advertised bandwidth is much higher, research has shown that the maximum application layer throughput of BLE is 236.7 Kbps [18]. Furthermore, some research has shown that under moderate error conditions application layer throughput drops even further, in their example to 58.48 Kbps [19]. This is further supported by [16], which shows that Bluetooth performance can drastically decrease under Wifi interference. Due to these factors it is difficult to choose Bluetooth LE if the reliabiliy of the connection can't be guaranteed.

Lastly, there is IEEE 802.11b, commonly known as WiFi. This is a widely used protocol, with high bandwidth and range capabilities. The bandwidth can go up to 100 Mbps, and advertised range is around 100m. A drawback of this technology is that the power consumption is higher compared to the other technologies. However, the power consumption is still low enough to reach 2 hours battery life on reasonably-sized batteries (e.g. on a 3.5Ah battery at 3.8V the battery life for the WiFi module would be around 130 hours).

Taking into consideration requirement G2P (The communication protocol must handle 9 concurrent signals.) and G6P.a (The device must transfer all signals lossless.) WiFi is chosen as the transmission protocol as it is deemed to be the most robust. Research supports the robustness of the WiFi protocol, Shin et al. demonstrating "Wi-Fi devices are scarsely affected by the presence of other wireless technologies operating concurrently" [16]. Given the large bandwidth, there is also a large margin before interference degrades performance to an unacceptable level. The same cannot be said for ZigBee and Bluetooth LE. Therefore, WiFi is the best choice of transmission technology.

4.1.6. ESP32

To transmit data using the WiFi protocol, of-the-shelf hardware is selected so that the project result complies with requirement G10P (The communication module must be available within the timespan.).

At the time of this project, there were not many options for a WiFi module. Due to high availability, the ESP32 was the best option by far. Figure 4.1[20] shows a functional block diagram of this device. According to the datasheet, ESP32 is a highly-integrated solution for Wi-Fi-and-Bluetooth IoT applications. The ESP32 implements both WiFi and Bluetooth. Although Bluetooth is not the chosen option for the transmission, there is no harm in having it on the prototype device to test with it if the time allows. On top of that, it has many peripherals which are interesting for pairing the transmitter and receiver.

Another benefit of the ESP32 is the big community surrounding it. Therefore it is easy to find documentation and debug the device. Next to that, the ESP32 is available in stock, so requirement G10P (The communication module must be available within the timespan.) can be fulfilled. Lastly, one of the team members happened to already have 3 ESP32's laying around which he kindly offered for use immediately at the start of the project, which allowed us to quickly start prototyping with the ESP32's before the hardware was fully designed. This time was necessary to fullfil requirement G10 (A prototype device must be functional within 10 weeks from the start of the project.)
As can be seen on the cover of this report, the ESP32-WROOM-32e is used. This device has the following relevant specifications.[21]

- ESP32-D0WD-V3 embedded, Xtensa dual-core 32-bit LX6 microprocessor, up to 240 MHz.
- 520 KB SRAM
- WiFi 802.11b/g/n
- Bit rate: 802.11n up to 150 Mbps
- Center frequency range of operating channel: 2412 - 2484
- On-board PCB antenna

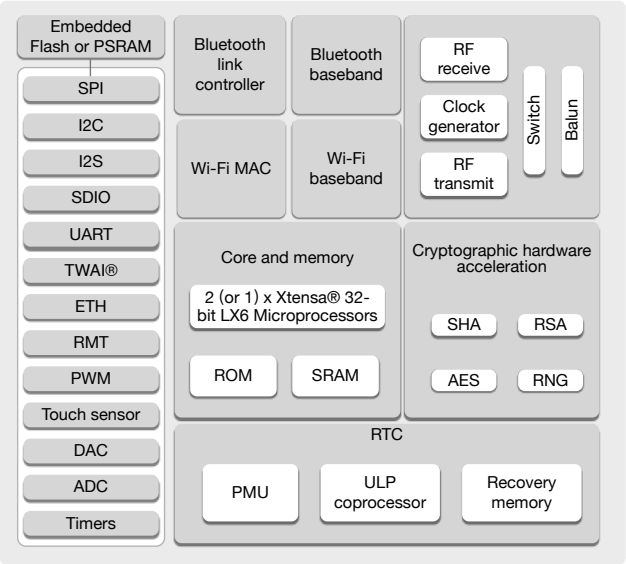


Figure 4.1: Functional block diagram ESP32

4.1.7. Compression

The main limitation of any wireless communication is bandwidth. Interference causes bit error, which effectively reduce the transmission rate. Therefore, a solution to be more resilient against interference is simply to require sending less data. A method of achieving this is by compressing the transmitted data before transmission. Since the chosen technology has a bandwidth much higher than is required, compression of the data is not necessary. However, if a protocol like ZigBee or Bluetooth is picked, then compression is an interesting solution to improve the reliability of the connection. To examine the feasibility of compression as a solution, an experiment analysing the compressibility of the signal data would have to be conducted. If the data is not compressible, then the solution is not useful, and only wastes computing power. In the case of using WiFi, compression is not necessary as bandwidth is

abundant. Therefore to save computing power (equivalent to battery life and heat output), compression is not implemented in the transmission protocol.

4.1.8. Handling corrupt packets

Since the data is being transmitted in an unstable medium, there is a possibility of bit errors, whereby the packet becomes corrupt. The corruption of packets is monitored by the transmission protocol using CRC codes, but this only tests the packet for corruption, and does not handle re-transmission of the packets. This feature needs to be designed and implemented. In general, there are three ways to handle a corrupt packet.

- **Null packet substitution** - When a packet corruption is detected, the packet is replaced with a "null" packet, which is a packet that represents a signal of all 0s. This solution is simple to implement, but creates artifacts on the output ECG signal that might confuse or annoy the caretaker.
- **Extrapolation** - Alternatively, when a packet corruption is detected, the previously received packet can be repeated. This is a simple solution to implement. Additionally, given that the BER is low enough, the artifacts produced by this solution might not be visible to the human eye, so it is transparent to the caretaker. Still, this solution involves distorting a medical signal, which is against requirement **G6P.a**.
- **Re-transmission** - A solution that produces no artifacts on the final signal is re-transmission. Here the corrupt packet is re-transmitted by the transmitter.

The only solution out of these options that matches requirement **G6P.a** is re-transmission.

Re-transmission

There are three main re-transmission strategies.

- **Stop and wait** - In the stop and wait re-transmission protocol, the sender will wait for the arrival of an ACK before continuing with the next packet.
- **Go-Back-N** - Go-Back-N is an example of a sliding window re-transmission protocol. A window of size N is maintained, and if a transmission error is detected (i.e. an ACK is not received on time), the entire window is re-transmitted. When an ACK is successfully received, the window slides forward.
- **Selective Repeat** - In selective re-transmission, similarly to Go-Back-N, a window is maintained. This window is sent in one go, and the receiver is expected to send an ACK for every packet. If an ACK for a packet has not been received by the sender on time, the sender re-transmits the packet.

Each of these protocols have different efficiency expressions. Efficiency is an expression that compares the transmission cost of the protocol as opposed to transmitting without re-transmission. The different re-transmission efficiencies are given by the following equations [22].

$$\eta_{S\&W} = \frac{1-p}{1+L_c} \quad \eta_{GBn} = \frac{1-p}{1+pL_c} \quad \eta_{SR} = 1-p \tag{4.1}$$

There is a tradeoff that these equations demonstrate, which is that more complex protocols have higher efficiency. Complex means that the protocol might incur higher memory usage, higher processing power or is more complicated to implement. Thus, to arrive to a decision, these factors need to be considered. Since the ESP32 is a powerful micro-controller, with enough RAM to support complex re-transmission protocols, the decision was made to implement the "Selective Repeat" protocol.

4.2. Pairing protocols

The application requires safe pairing between receiver and transmitter according to requirement G4P.b (The device must ensure safe connection between transmitter and receiver.) This means that the receiver and transmitter need to find each other when the devices are turned on. This needs to happen fast, according to requirement G5P.a (The pairing of the device must work within 2 seconds.)

Figure 4.2 [23] shows an extensive list of pairing methods.

	Device/Equipment Requirements		User Actions			
Pairing Method	**Sending Device**	**Receiving Device**	**Phase I: Setup**	**Phase II: Exchange**	**Phase III: Outcome**	**OOB Channels**
Resurrecting Duckling*	Hardware port (e.g., USB) on both and extra cable		Connect cable to both devices	**NONE**	**NONE**	Cable
Talking to Strangers*	IR port on both		Activate IR on both & find /align IR ports	**NONE**	**NONE**	IR
Visual Comparison: Image, Number or Phrase	Display + user-input on both		**NONE**	Compare two images, or two numbers, or two phrases	Abort or accept on both devices	Visual
Seeing is Believing (SiB)*	Display + user-input	Photo camera + user-output	Activate photo mode on receiving device	Align camera on receiving device with displayed barcode on sending device, take picture	Abort or accept on sending device based receiving device decision	Visual
Blinking Lights*	LED + user-input	User-output + Light detector or video camera	Activate light detector or set video mode on receiving device	Initiate transmittal of OOB data by sending device	Abort or accept on sending device based receiving device decision	Visual
Loud & Clear ▪Display-Speaker ▪Speaker-Speaker	User-input on both + ▪display on one & speaker on other, or ▪speaker on both		**NONE**	Compare: two vocalizations, or display with vocalization	Abort or accept on both devices	▪Audio, or ▪audio + visual
Button-Enabled (BEDA) ▪Vibrate-Button* ▪LED-Button* ▪**Beep-Button***	User input + ▪vibration , or ▪LED, or ▪beeper	User output + One button +	Touch or hold both devices	For each signal (display, sound or vibration) by sending device, press a button on receiving device	Abort or accept on sending device based receiving device decision	▪Tactile, or ▪Visual + tactile, or ▪Audio + tactile
Button-Enabled (BEDA) ▪Button-Button*	One button on both + user-output on one		Touch or hold both devices	Simultaneously press buttons on both devices; wait a short time, repeat, until output signal	**NONE (unless synch. error)**	Tactile
Copy–and-Confirm*	Display + user-input	Keypad + user-output	**NONE**	Enter value displayed by sending device into receiving device	Abort or accept on sending device based on receiving device decision	Visual
Choose-and-Enter*	User input on both devices		**NONE**	Select "random" value and enter it into each device	**NONE (unless synch. Error)**	Tactile
Audio Pairing* (HAPADEP variant)	Speaker + user-input	Microphone + user-output	**NONE**	Wait for signal from receiving device.	Abort or accept on sending device	Audio
Audio/Visual Synch. ▪**Beep-Beep** ▪Blink-Blink ▪Blink-Beep	User-input on both + ▪Beeper on each , or, ▪LED on each, or ▪Beeper on one & LED on other		**NONE**	Monitor synchronized: ▪beeping, or ▪blinking, or ▪Beeping & blinking	Abort on both devices if no synchrony	▪Visual, or ▪ Audio, or ▪Audio + visual
Smart-its-Friends*, Shake-Well-Before-Use*	2-axis accelerometers on both + user-output on one		Hold both devices	Shake/twirl devices together, until output signal	**NONE (unless synch. error)**	Tactile + motion

***Resistant to a "rushing user" behavior**

Figure 4.2: Summary of pairing methods

Pairing methods commonly supported by state-of-the-art protocols and devices can generally be categorized into three types.[24] All pairing methods shown in 4.2 fall into one of these categories.

- "Just works" pairing method. This method does not use any form of authentication. It is common for devices that have no user interface (UI). This method is vulnerable for a man-in-the-middle attack.
- When a UI is available, somewhat more reliable pairing is possible. This is mostly done with a random passkey generated by one of the devices, which should be verified on the other device by

the user. This is reasonably safe, but still has no defense for e.g. shoulder surfing. Next to that, it is inconvenient for the user.

- The third type is out-of-band (OOB) pairing. This method uses a near field secondary channel which should be protected from eavesdropping. This can be a physical connection or e.g. NFC.

Option 1 is not a reasonable option for this application, as the pairing should be robust and it should not be possible for the transmitter to connect with anything other than the dedicated receiver. Option 2 is very inconvenient in life threatening situations because it takes too much time from the ambulance personnel. OOB pairing is the best option, so next sections are devoted to different OOB pairing. One should note that strictly speaking, a user that takes part in the pairing is also OOB. However, for now we state that an OOB channel does not include the user.

4.2.1. SYNCVIBE

Lee, Kyuin, et al [24] proposes a improved pairing method which uses physical vibration. It is called SyncVibe. It makes use of a vibration motor and and accelerometer. It is originally designed and tested for smartphones and wearables. The user makes physical contact between the two devices and the pairing is automatically done. The average pairing time with this method is 6.74 s for 150 bit pairing information. This allows for unidirectional pairing. This method does not fulfill requirement G5P.a (The pairing of the device must work within 2 seconds.)

4.2.2. Physical contact

Stajano and Anderson [25] discuss the security issues related to pairing methods using short range wireless channels. They advocate that in many applications, physical contact is the best option. Pairing bits are shared over a conducting medium, making it safe, fast, cheap and simple. There is no need for cryptography and there is no ambiguity about which two devices are involved in the pairing.

As there is no downside with using a hardware port to pair transmitter and receiver, this method is chosen for pairing the devices.

4.2.3. Protocol physical contact

The transmitter and receiver will be connected to each other with metal contacts. It is important that this is waterproof, especially when the transmitter and receiver are not connected to each other. This can be accomplished by waterproof contacts that close when disconnected. Another way is using a protocol/circuitry that allows all lines to be shorted without implications. The speed of the protocol is not really of interest, as the amount of data that should be transferred to pair are mere bytes (6 bytes for MAC address and 16 bytes for encryption key[14]). Table 4.2 [26] shows a summary of common wired protocols.

	UART	CAN	USB	SPI	I2C
Pros	Well known Simple	Secure Fast	Secure Fast Plug and play	Fast Lowcost Universally accepted Large portfolio	Simple Plug and play Cost effective Universally accepted
Cons	Limited functionality Point to point	Complex Limited portfolio Automotive oriented	Power master required No plug and play software Extra drivers required	No plug and play hardware No fixed standard	Limited no. of components due to capacitance effect

Table 4.2: Summary of common wired protocols

Another protocol not listed in the table is a one-wire protocol. This protocol uses only one wire to communicate and a ground. These two wire connections allow for a very simple connector.[27]

4.2.4. Comparison

The one-wire protocol seems like an interesting solution. Upon further research, it seems like this type of protocol is mostly used to power and communicate with large amounts of small sensors.[28][29][30][31][32] Its usage might be extended to pairing, but this seems very experimental. In view of requirement G10 (A prototype device must be functional within 10 weeks from the start of the

project.), this protocol is not chosen.

UART is especially made for communication between 2 devices only. It uses 2 wires (+ground). It includes a start and stop bit and a parity check. Both devices need to be set at the same BAUD rate.

CAN is not widely used, mostly in automotive. It supports communication over long distance and is mostly used when multiple devices are on the line. It is a peer-to-peer network and harder to implement.

USB is fast and safe, but needs a powerful master with extra software and drivers. It has many wires and has a standardized connector.

SPI does not support acknowledgements of received data. This makes it not very useful as the data transferred should arrive by all means. It is mostly used on PCB with components which are permanently connected. It can still be used, but then it needs an extra layer on top of it for acknowledgements.

I2C uses 2 bidirectional wires. It has acknowledgements to ensure safe communication.

UART and I2C are both widely used and most microcontrollers support both types of communication. I2C has the preference, as it makes use of acknowledgements, which increases reliability.

4.2.5. Confirmation

Even though pairing should be robust and never fail. Users should quickly be able to see whether the transmitter and receiver are paired in case of an error. This is stated in requirement G4P.c (The pairing must be easy to confirm.) Therefore, both devices need a visual or audible output so a user can confirm the pairing. Not only should the user be able to see if the devices are paired, but also that they are without a doubt paired with each other. The (unlikely) event of the receiver being connected to an unwanted transmitter without the knowledge of the user is unacceptable in case of the ECG application. A way of confirmation can be accomplished in multiple ways:

- A LED on both devices. These LEDs can flicker synchronously to each other. However, there is still the risk that 2 devices just happen to flicker synchronously even though they are not paired.
- An RGB LED on both devices that show the same color when paired. However, in a rush the user might wrongly judge two colors to be equal.
- A numeric LED display on both devices that show the same (randomly generated) number. This is the safest visual option.
- A button on one device, and a LED on the other device. The LED will turn on when the button is pressed and the devices are paired. The downside is that the user is obliged to pick up the device to see if they are paired. Next to that, there is no indication if the devices happen to disconnect.
- A button on one device, and audible buzzer on the other device. The benefit of this is that the user does not have to focus on a visual output and both devices do not have to be in visual sight. On top of that, hearing the sound can easily become a habit for the user.

In the end, all above options are vulnerable to hasty user behavior. The only option is to have the devices make a sound when they are NOT paired, but this will clearly lead to annoyance of the user. Even though this seems trivial, it can lead to the user getting creative in bypassing this important security measurement, which is something to be avoided.

The best option seems to be the button and audible buzzer in combination with an LED. Both devices have a button and a buzzer in combination with an LED. These give both an audible and visual confirmation of the pairing when the button on the other device is pressed. This buzzer can also indicate when the device happens to unpair. Because of time constraints of this group, the software is only designed for one button on the transmitter and one button on the receiver. The hardware is designed for both devices to implement button and receiver. They will be used in a later stage of the project.

4.2.6. I2C waterproof

When the I2C connectors are submerged in water, they run the risk of a low-impedance connection to each other. In the worst case there is a low-impedance connection from ground to SCL or SDA, which

would cause a connection from the 4.7 $k\Omega$ pullup resistor to ground. Using $P = \frac{V^2}{R}$, the power dissipated in the resistor is 2.3 mW. So, even with a continuous short there won't be damage to the equipment.

However, water does not cause straight shorts, it adds DC resistance of its own. An independent measurement performed by the subgroup showed that a resistance of 250 $k\Omega$ can be expected. This means that the pulldown due to water is significantly weaker than the 4.7 $k\Omega$ pullup. If the SDA and SCL line are shorted to each other, nothing happens as they are on the same voltage. In reality, they will also have this 250 $k\Omega$ impedance between them, making sure the pulldown is always significantly stronger. Therefore it is expected that this pairing method functions underwater.

5 Implementation

The implementation chapter describes the choices made and techniques used during implementation of the design. Each section discusses a particular aspect of the implementation.

The whole implementation is done on an ESP32, of which the documentation [33] and datasheet [20] are the main sources of information. The implementations were first done on two ESP32 prototype boards which allowed the group to start immediately and do some quick prototyping before the hardware was designed. When the hardware is designed and made, the software is transferred to the ESP32 chips on the PCB. As both receiver and transmitter use the same concepts, the following implementations are used for both devices unless stated otherwise.

The ESP32 is programmed with the Espressif IoT Development Framework. It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++. [34]. The ESP32 comes with FreeRTOS as kernel. FreeRTOS is specifically designed for microcontrollers. The FreeRTOS reference manual[35] and the FreeRTOS kernel documentation [36] are used as main source for implementation.

Snippets of the code can be found in Appendix A.

5.1. Stability

Reliability is an important requirement for the project. One of the modules most vulnerable to unreliability is the software implementation of data transmission. Complexity is the main cause of bugs (and therefore instability). Thus, the implementation phase should always use the simplest working solution. Unfortunately, complexity naturally arises when interacting with pre-written libraries on the ESP32-IDF platform. Nevertheless, throughout the implementation phase attention is paid to reducing complexity where possible.

5.1.1. Error Conditions

As with every software project, the implementation will naturally encounter error conditions. Some of these errors are recoverable, and the implementation is expected to resolve recoverable errors without interruption of the transmitted signal. However, some errors are unrecoverable, and for these errors the implementation is not expected to resolve them. These unrecoverable errors are identified, and summarised in the following list.

- **Transmission link completely drops** - If the WiFi transmission link between the receiver and transmitter completely drops, then there is no possibility of transmitting a signal on time.
- **Buffer takes too long to send** - The implementation keeps a buffer of 0.5s worth of ECG samples. Under stable operating conditions these samples can be transmitted in under 0.5s, meaning they are on time and there is no loss of information. In the case that the transmission link quality degrades such that the samples cannot be transmitted within 0.5s anymore, there is an unrecoverable error, since the system can no longer deliver the data on time.

- **Physical link errors** - In multiple places the system uses I2C to communicate between microprocessors. If for any reason the physical link between the systems are unexpectedly destroyed, there is an unrecoverable error.
- **Other hardware errors** - The ESP32 can theoretically encounter hardware errors, which is outside the scope of what software can recover from.

The most sensible method of dealing with unrecoverable errors is to inform the user as quickly as possible of the failure. An unrecoverable error means loss of signal information, and it is the user's choice how to act in such a situation. The user might simply choose to restart the device, but in a medical emergency, a physician has many choices in case of failing equipment. The user is informed of the failure of equipment by a continuous buzzer and an LED.

5.2. Parallel processing

The ESP32 has multiple functions. The main functions are stated in the following list.

- Pairing receiver and transmitter.
- Sending/receiving ECG-data via WiFi.
- Communicating ECG-data with the DSP chip via I2C.

All functions should be run in parallel. Even if the device is already transmitting data, the pairing mechanism should still be active for if the devices should connect to another device. The I2C communication should function, independent of the state of WiFi communication. Therefore, a concept called parallel processing is implemented. FreeRTOS allows us to multitask and use both processors of the microcontroller in an efficient way. First some general information is given about the FreeRTOS kernel. In Section 5.2.2 and Section 5.2.3 there are explanations on how tasks and queues are used to implement the wanted functionality.

5.2.1. FreeRTOS

FreeRTOS is a real-time kernel on which applications can be built. It uses separate threads of execution called tasks. Only one task at a time runs per processor. The other tasks are on hold. Figure 5.1 visually shows this concept.

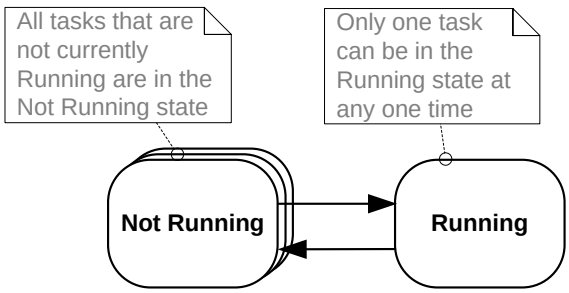


Figure 5.1: Top level task states and transition

The tasks are swapped in and out of the running state by the FreeRTOS scheduler. There are three reasons for a task to be in the Not Running state which are listed below.

- The task is in a blocked state. This means that the task is waiting for 2 different types of events. The first one being time related events. The task is waiting for a certain amount of time or waiting for an absolute time being reached. The other event a task can be waiting for is a synchronization event. This event can come from another task or an interrupt.
- A task of higher priority is running. Each task is given a priority on creation. When two tasks are waiting to be swapped into the running state, the scheduler chooses the tasks with the highest priority.
- A task of equal priority is running. When two tasks have equal priority, the task scheduler swaps them in and out alternately, giving them equal processor time. These swaps can only happen at a tick interrupt, of which the frequency is by default 100 ticks per second.

FreeRTOS has different possibilities of creating synchronization events. One of them is a concept called Queue. A queue can hold a finite amount of fixed size data items. By default a queue is used as First In First Out (FIFO) buffer. FreeRTOS uses the queue by copy method, this means all bytes placed in the queue are copied. This way, the data keeps existing if the original data is removed.

Queues are usually written to from multiple tasks and read from one task. A reading task can be in a blocked state waiting for data to appear in the queue. A writing task can be in a blocked state when for example the queue is full, waiting for it to empty.

5.2.2. Task

The transmitter and receiver both have four tasks running. They are discussed separately with the help of flowcharts.

Pairing

This task is called `i2c_task` in the code. Both receiver and transmitter have an I2C port on the outside of the device. When these are connected to each other, the MAC-address and encryption key are swapped.

When the user connects the devices together, the connection is really unreliable at the start. For example, the data and clock lines can be connected, while ground is not. It is also possible that the lines are connected for only a brief period of time. These abnormalities can make the system really unstable, if it is not designed for it. Therefore, the pairing is implemented as the state machine shown in Figure 5.2. The left and right side of the figure show receiver and transmitter respectively. The receiver is the I2C master and the transmitter is the I2C slave. There is no particular reason for the division of master/slave relation. The data transferred is symmetric, so the master/slave can be swapped without problems.

The state machine comprises of many error handling and checks to confirm the pairing. The master continuously (with an interval of 20ms) sends an empty message to the slave. This time delay is there to prevent this task from being in the running state all the time. This is to save power consumption. The master is now waiting for a slave to be connected and respond with an acknowledgement. When the presence of the slave is confirmed, the master writes the MAC-Address, encryption key and a calculated cyclic rendundancy check (CRC) code to the slave. The slave confirms this and checks whether the CRC matches with the received data. When there is a mismatch, the slave knows that there has been a failure in the transmission and sends back an error to the master. Both master and receiver now reset and the cycle is repeated.

If the CRC matches, the slave sends back it's MAC-address to the master which also checks if the CRC matches. When the addresses received by master and slave are different from the ones already known, an event is queued that a new device is connected.

All functions have a timeout implemented of 1 second. When something unexpected happens with the communication and nothing happened on the line for 1 second long, both devices reset to the first state and try again. After most types of errors, the devices clears either the TX-, RX- or ring-buffer (Section 5.3.1 to make sure a new attempt of connection is made with empty buffers.

The source code for both transmitter and receiver can be found in Appendix A.1 and A.2 respectively.

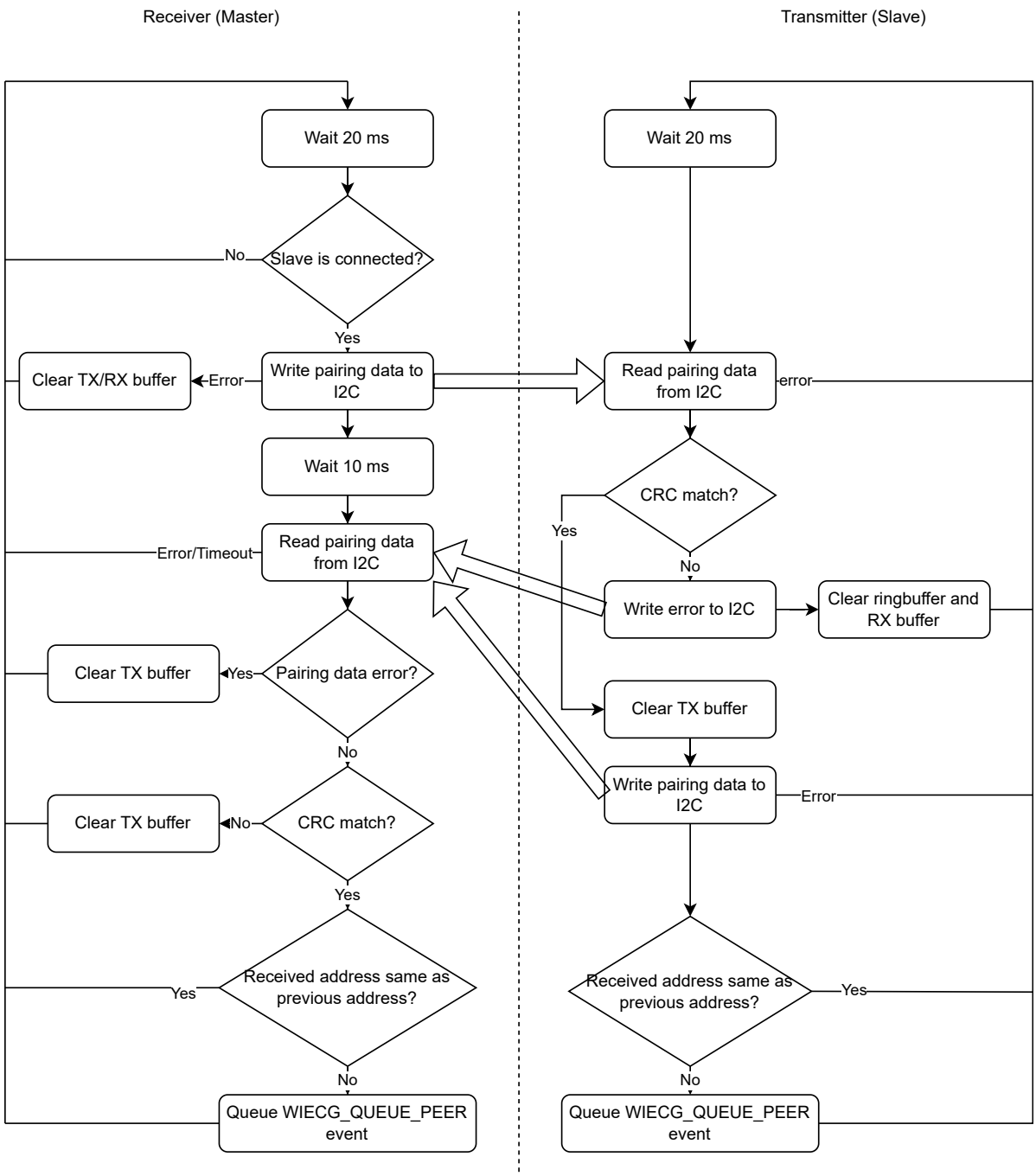


Figure 5.2: Flowchart of pairing task

Sending/receiving ECG data over WiFi
This task is called `espnow_task` in the code. This task is responsible for handling the WiFi communication. The flowchart of this task is shown in Figure 5.3.

In the transmitter, if there is no device paired, all messages that are queued to be send are cleared. There is simply no device to send it to. It then gets an event from the ESPNOW queue which can be of four different types. The first option is that there is received data in the queue. This is most likely an ACK message form the receiver. The second option is that the WiFi module is ready to send new data. A flag is now set indicating this. The third is that the sending timer was triggered. This is an event queued by the pairing task. The last option is that a new device is paired. This task will then take care of configuring this new device. If the previous indicated flag is set, it will check if there is data in the

message queue. If there is any, it will now send data from the message queue over WiFi and set this flag false again. If there is no data in the message queue, it will get ECG data from the buffer and performs the same operations.

In the receiver, this task waits for an event in the ESPNOW queue. If new data is received, this task first checks if the device is paired. This prevents the receiver from processing ECG data from an unpaired (random) device. The data received can be of two types. It can be ECG data, which will be queued so it can be forwarded to the DSP microcontroller. It can also be button data, indicating that the button is pressed on the transmitter and the buzzer needs to be turned on or off, depending on whether it is a press or release.

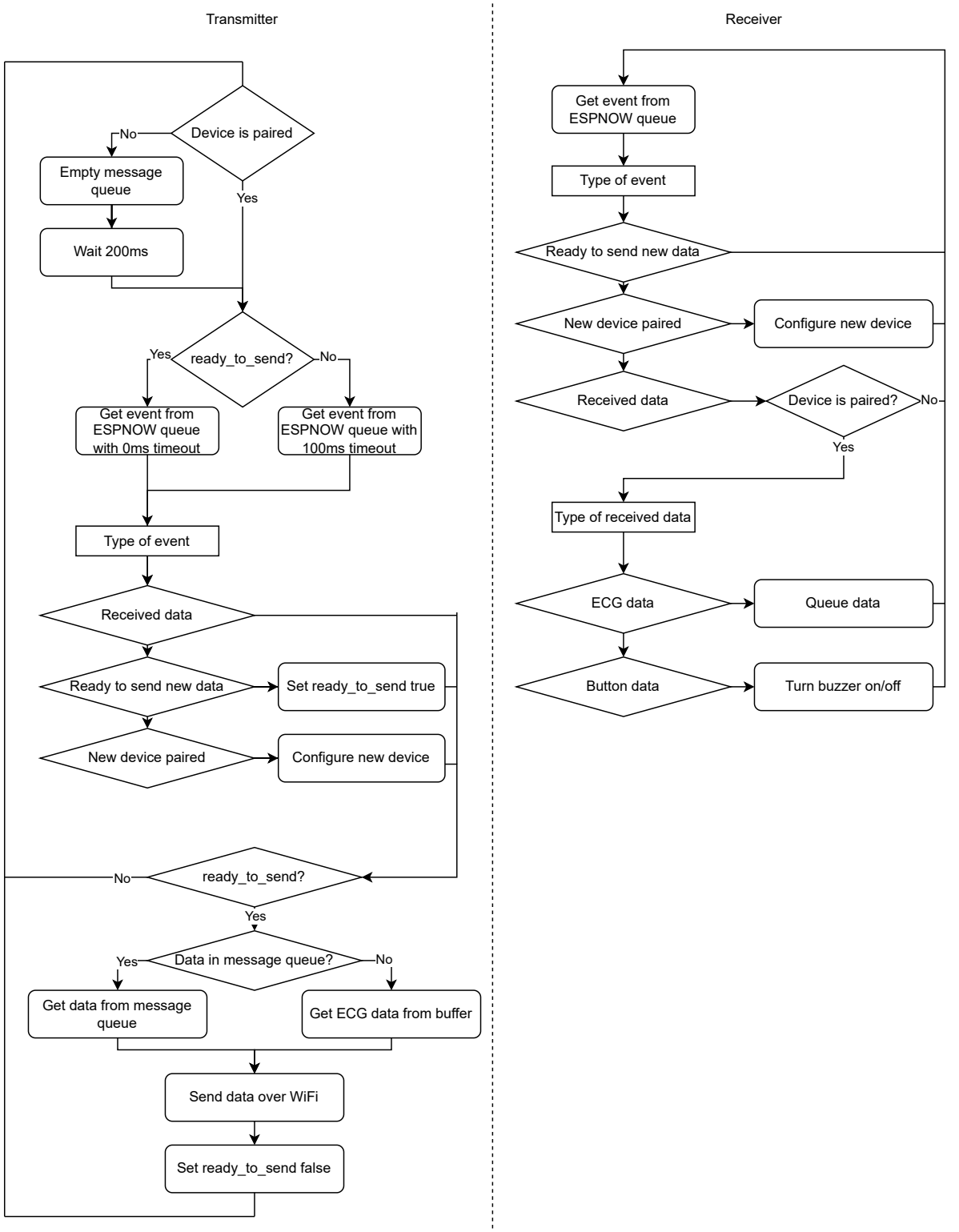


Figure 5.3: Flowchart of the WiFi task

The source code for both transmitter and receiver can be found in Appendix A.3 and A.4 respectively.

I2C communication for ECG data
This task is called `i2c_task` in the code.

The communication with the DSP microcontroller is also done with I2C like the pairing. The implementation is simpler because the connection is known to be reliable (there is a permanent connection) and the data only goes one-way, except for the acknowledgements.

The ESP32 in the transmitter only receives data from the DSP microcontroller. Only the master can initiate a transfer. Therefore, the ESP32 is configured to be the slave. The tasks simply waits for ECG-data to be received and queues an event with the newly received data.

With the same reasoning the ESP32 in the receiver is configured to be the master. This task pulls data from a queue where the received ECG-data is stored and sends it over the I2C line to the DSP microcontroller.

Idle task
This task is part of FreeRTOS and is always active. It has the lowest priority and has no function whatsoever. However, it needs to be run every once in a while, as this task resets the watchdog timer. In other words, if other tasks take up 100% of the processor, the system resets after a couple of seconds.

5.2.3. Queues
The transmitter has the following queues implemented.

- `espnow_queue`. This queue holds all the events regarding the WiFi communication. This queue is mostly written to from the ESPNOW callbacks. The pairing task also writes to this queue when a new device is paired. The WiFi task (`espnow_task`) reads from this queue.
- `message_queue`. This queue holds all messages that are ready to be sent. This queue is mostly written to from the task that receives I2C data from the DSP microcontroller (`i2c_task`), which writes ECG data to this queue. When the button is pressed/released, an interrupt places a message in this queue about the status of the button.

5.3. Communication
The following section discusses the implementation details of the different communication technologies utilized.

5.3.1. I2C
The ESP32 includes dedicated I2C hardware and software support. To initialize the I2C driver, one needs to choose which pins to have the SCL and SDA lines. These can be any GPIO pin on the ESP32. This is encoded in structure passed to `i2c_param_config()`. Initialization is finalized using `i2c_driver_install()`. Communication on the master and slave side differs, and are thus split up into different sections.

Master
When the ESP32 is acting as master, the steps for communication are as follows. Firstly, a command sequence is initialized using `i2c_cmd_link_create()`. This function returns a `i2c_cmd_handle_t`, which acts as a buffer in which I2C commands can be stored. Next `i2c_master_start()` is called, which queues a START condition (the master pulls SDA low while SCL remains high). At this point all slaves on the bus are listening, and the master needs to write a slave address to the bus to select a particular one it wishes to communicate with. This is accomplished using the function `i2c_master_write_byte()`, which queues a command to write a single byte to the bus. Once the slave is selected, the master writes/reads the bytes of the message, which is accomplished using the function `i2c_master_write()` or `i2c_master_read()`, which both take as argument a buffer and buffer size, writing/reading the buffer entirely, respectively. To finalize the communication, the master enqueues a STOP condition, accomplished using the function `i2c_master_stop()`. This completes the construction of the command queue. The command queue is executed using `i2c_master_cmd_begin()`.

Slave
The slave communicator only has two main functions it can use to communicate with its master. Namely, `i2c_slave_read_buffer()` and `i2c_slave_write_buffer()` for reading and writing, respectively. The slave has a built-in hardware ringbuffer for storing the out-bound and in-bound data. Data is copied out of this

ringbuffer by the ESP-IDF toolkit automatically, using interrupts, and placed into a larger userspace buffer. The latter buffer is where the `i2c_slave_*` functions read and write to.

5.3.2. ESP-NOW

Figure 5.4 shows the high level overview of the execution flow of the ESP-NOW subsystem. The execution starts by initializing ESP-NOW on the ESP32. To perform initialization, first WiFi is initialized. On the receiver side, the Access Point interface is used, which means that extra attention must be paid. By default, this Access Point is exposed as a conventional WiFi network, leaving the possibility for other WiFi devices to connect to the receiver ESP32. The ESP-IDF API does not allow this functionality to be disabled. Hence, to ensure adequate safety, the access point's name is randomised, and hidden (meaning it doesn't advertise itself to other WiFi devices), and its password is randomised as well. The name and password are renewed every boot cycle.

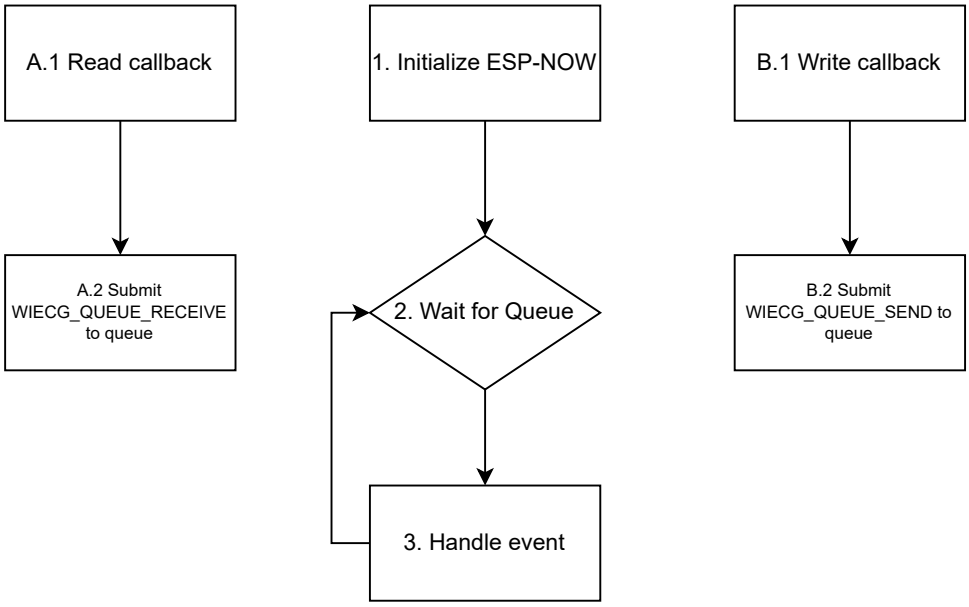


Figure 5.4: High level overview of execution flow of ESP-NOW subsystem

Once ESP-NOW is initialized, a task is created which waits for events to be pushed to a queue. Events can be pushed to the queue using "callbacks". These are functions registered to ESP-NOW that are called to signal a transmission event. There are two callbacks:

- **A.1 Read callback** - The read callback is called by ESP-NOW whenever there is data ready to be read. The callback function will copy the read data into a buffer, and submit it to the queue for further processing.
- **B.1 Write callback** - The write callback is called by ESP-NOW whenever the interface is ready to send more data. The callback function submits an event to the queue to signal more data can be sent.

These callbacks are run from the ESP-NOW internal WiFi task. Performing complex processing or blocking operations in the callbacks will therefore also block the WiFi task from running. Therefore, the callbacks only submit some work to a queue, where more complex processing is performed instead.

5.4. Data structures

5.4.1. Bitmap

To implement the Selective Repeat re-transmission protocol, an efficient data structure for keeping track of which packets were ACK'd is required. In the implementation a bitmap was used. The ESP32 is a 32-bit processor, so the implementation uses an array of `uint32_t` integers to store a bitmap. The bitmap

is initialized using `wiecg_bitmap_init()`, which takes as parameter a size. This parameter is the number of bits that need to be stored, at a minimum. Internally, this number will be rounded up to the nearest multiple of 32 (as a `uint32_t` stores 32 bits).

Bits can be set and unset using `wiecg_bitmap_set()` and `wiecg_bitmap_unset()` respectively. The status of a single bit can be checked using `wiecg_bitmap_get()`. The full API can be found in Appendix A.5.

An ACK accompanies the sequence number of the packet it ACK's. When the transmitter receives an ACK, it uses this sequence number as an offset into the bitmap, and sets the according bit. After a timeout of 100 ms, the transmitter checks the entire bitmap. If it encounters an unset bit, it re-transmits that individual packet. The bitmap is a space efficient data structure for accomplishing the semantics required for Selective Repeat.

5.4.2. Associative array

Another data structure required for Selective Repeat is a buffer that can act as a window. To design this data structure several requirements were considered.

- **Packets can arrive out of order** - Some packets can never arrive, and after re-transmission they should still be processed in order. Thus the data structure must be able to handle inserting packets in order, and not strictly sequentially.
- **Receiver: I2C transmission** - The data that is received is transmitted again over I2C. At the point of I2C transmission the data must be in order and complete.
- **Transmitter: I2C source** - The data that is transmitted by the transmitter arrives from the DSP module over I2C. There must be a complete window available to transmit, before transmission commences.

Considering all of these requirements, the decision was made to implement a doubly-buffered associative array. An overview is shown in Figure 5.5. The data structure consists of two flat arrays. At any point in time exactly one array is being written to, and only one is read from. The write array is addressed using modulo-arithmetic, so if the buffer size is n then writing to location i writes to location $i \mod n$. The buffers can be swapped when the writing buffer is full.

On the transmitter side, this array is used to receive packets from the I2C of the DSP micro-controller. They are written to the write buffer. The read buffer is used as the window for the wireless transmission. Once the write buffer is full, the bitmap is checked. If all of the bits of the bitmap are set, then the buffers are swapped.

On the receive side the array is used to receive packets from the transmitter over WiFi. Because of the modulo-arithmetic property of the array, the packets arriving out of order can be placed in the receive array in $O(1)$.

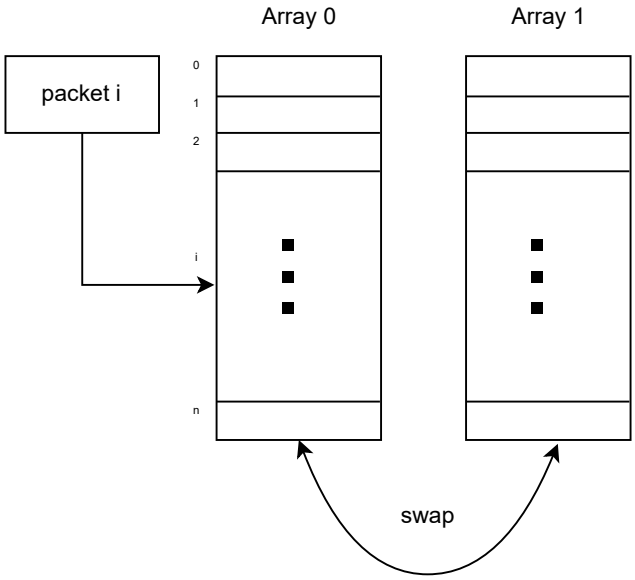


Figure 5.5: Design of the associative array

5.5. Re-transmission

Re-transmission is implemented using the aforementioned data structures. Additionally, a timer is used. The timer is armed after each window is transmitted. When a re-transmission is performed, the timer is armed again. The timer is armed a maximum of ten times before there is unrecoverable data loss. In the future this will be determined based on time elapsed, instead of simply ten retries. Whenever the timer is triggered, it submits an event to the `espnow_queue`. The event is received in the WiFi task, where it checks the bitmap for any packets requiring re-transmission. Appendix A.6 shows the snippet of code that performs the bitmap check and re-transmission.

5.6. Button debouncing

A problem that was encountered when using the button was switch bouncing. This issue is not directly related to the functionality of the ECG-device, but is something to be solved nonetheless. This is the phenomenon that when the button is pressed or released, it quickly bounces between contact and non-contact a couple of times before settling in its final state. This can be solved with hardware (debounce circuits) and with software. Both options have their pros and cons. It was chosen to tackle this problem with a software-based solution. The main reason was that the hardware was already designed, so it required either a redesign or an add-on. This was determined to be too much effort for a secondary project goal. If the bouncing switch problem would have been considered from the beginning, the same decision would have been made. The extra hardware would have required extra space on the PCB, which is unnecessary as it is a problem than can be solved with software just fine.

When the button changes state, an interrupt service is called. This interrupt either starts a timer or resets a timer if it was already running. When this timer reaches 50ms, another interrupt service is called, reading the final state from the button. This is enqueued to be send to the other device. The 50ms is determined experimentally, by increasing this delay until no bounces were observed anymore. In other words, the state of the button is only enqueued when the timer hasn't been reset for the past 50ms. This eliminates bouncing, as the button has been in a stable state for 50ms.

6 Measurements

In this chapter some measurements are done of the basic functionality of the device. The system still needs more elaborate testing. More about this can be read in Chapter 7.

6.1. Data rate

In this section, the data rate from transmitter to receiver is measured and plotted against the distance. The test is performed in the following way.

- The devices are paired.
- The transmitter fills a buffer with 500 (of 9 signals) random data samples.
- The transmitter starts a timer.
- The data is transmitted at a maximum rate.
- If data packages are lost, retransmissions are performed.
- The transmitter stops a timer. This will only happen when all packages have been acknowledged.

In this test, a sample size equivalent to 1000 milliseconds is send. The result of the test gives the time it takes the devices to transfer this data. It is first tested on modules of the final prototype. Due to poor results, it is then tested on two ESP development boards.

6.1.1. Modules

The ESPs are soldered on a PCB with a ground plane underneath the antenna. The performance of the antenna is influenced by this ground plane. Tests show that the orientation also influences the performance of the device. The following test is executed in a lab with many people with phones, so the WiFi interference is considered to be relative high. With the ESPs facing each other (with a line of sight between them) the devices work up to a distance of 5 meters with a transmission time of 430 ms. When the transmitter is flipped, so that the ground place is between the two ESPs, the transmission stops working at 1 meter.

More about this problem of the ground plane can be read in the hardware report[2].

6.1.2. ESP development boards

The test is done on two ESPs which were used for rapid prototyping. This is done due to the problems with the ESPs on the modules. Regarding the positioning of the antenna, the test on these devices can be seen as an ideal situation.

A test is performed under the following circumstances.

- The receiver is placed at the beginning of a hallway in an apartment building with many other WiFi sources.
- The transmitter is moved away from the receiver through the hallway.
- An average measurement of 5 buffers is taken per meter.

The results of this test are shown in Figure 6.1. It can be seen that the transmission time is around 250ms and that there is no significant decrease or increase in the transmission time. Although this is not shown in the figure, this trend continued up to 40 meters, which was the end of the hallway. To continue testing until failure, the stairs were taken. These stairs were around 3m away from the receiver. One floor down resulted in an average transmission time of 1266ms, two floors down resulted in complete failure of transmission.

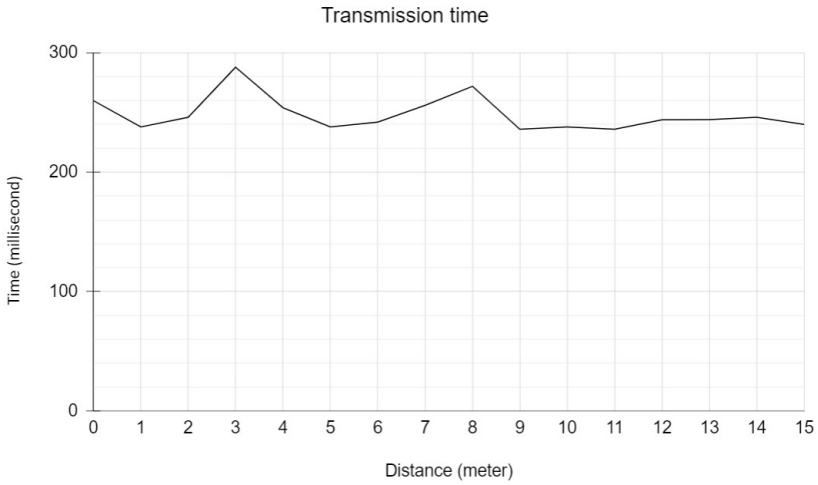


Figure 6.1: Data rate measurements plotted against distance

As stated by requirements G2P, G6D.a and G6D.b, the data link should transfer 9 signals of 16 bits at a rate of 500 Hz. This comes down to an effective rate of 72kbit/s. In line of sight circumstances, 1000ms of data is send in about 250ms. From this information, it can be extrapolated that the maximum effective data rate is 288 kbit/s.

6.2. Pairing

During implementation and testing, problems were encountered with the pairing. More about this can be read in Chapter 7. When both devices were reset before pairing, it never failed and paired instantly when the wires are connected. However, when one of the devices was already on (and paired to a device) it sometimes failed to pair.

7 Discussion

In this chapter, the achieved results will be discussed and interpreted and possible improvements will be proposed. The final prototype of this subgroup is yet to be integrated in the complete system, but it works as standalone now with some problems that still need to be solved.

Data rate

Actual data rate achieved was much lower than expected beforehand. The measured effective data rate is 72 kbit/s, which is much lower than expected. WiFi was chosen as a solution that would be more than capable of achieving high data rates. Experiments were done with manually changing the data rate of the ESP32 chips, but this did not result in a measureable change in effective data rate. This suggests that the bottleneck is not in the wireless solution, but rather some implementation inefficiency.

Processor usage

The ESP32 is chosen for flexibility. It is a quite strong device. Tests need to point out whether the processor is fully used or that the same application could be made on a device with less computation power. One way to measure this is to measure the amount of time the idle task is scheduled. If the idle task runs a majority of the time, it means that the ESP32 is underutilized.

Pairing

I2C was not as stable as expected and required a lot of band-aiding. Sometimes the device requires a reboot for it to pair, which is unexpected behavior. Most likely this is due to programmatic error. Due to the asynchronous connection of the I2C connection, the ESP32 often transmitted only part of a message. This caused the hardware ring buffer to be filled with partial messages, instead of complete ones, causing repeated invalid messages to be transmitted. Flushing of the hardware buffers was attempted, however, to little success. Given this difficulty, exploring other protocols is also a sensible next step. When pairing succeeds, it succeeds almost instantly.

Safety

The communication is implemented with ESP-NOW, which supports encryption. Further research needs to be done about the safety of this protocol and whether it is good enough for transferring medical data.

Latency

Currently, the device is implemented with a buffer size equivalent to 0.5 seconds. It was assumed that this delay would not be a problem for the ambulance personnel. This assumption should be verified. Increasing this buffer would increase the reliability of the system, because it can cover up a longer duration of lost signal.

Future measurements

While having performed transmission time vs distance measurements, there are still many important experiments left. An extensive but not exhaustive list is given of future measurements.

- **Interference testing** - So far there has not been any limit testing in terms of interference. A good experiment would be to continuously add more interference to the environment around the devices. This can be achieved in several ways, for example continuously adding more WiFi enabled devices into a room. Another method of testing would be to use a commercial interference tester.
- **Ambulance testing** - The transmission system should also be tested in the intended environment, which has not been done so far. This means that the system should also be tested inside of an ambulance, which provides a different interference pattern than an apartment complex.
- **Temperature measurement** - Another measurement which is necessary is to perform temperature measurements of the system under load. It needs to be ensured that the devices are operating within their thermal limits, under all circumstances. Variations of this test could be testing at different ambient temperatures, so that it can be ensured the devices still work in different climates.
- **Pairing speed** - While qualitatively pairing is instant, there should still be a rigorous measurement of the exact amount of time that it take to pair two devices.
- **Pairing fail amount** - There should be an experiment in which the failure rate of pairing is empirically determined. This experiment can be done by repeatedly pairing devices and recording when this process fails.
- **Outdoor testing** - The devices have not been tested outdoor, but the transmission reliability and rate should also be tested outdoors. The indoor distance experiment can be repeated outdoors to perform this measurement.
- **Underwater testing** - The devices were not tried underwater, mainly because there were only non-waterproof prototype boards available. However, this should still be tested once a fully complete waterproof design is available.

8

Conclusion and recommendation

8.1. Conclusion

Concluding, this thesis described the process of designing and implementing a prototype wireless protocol for the WiECG project. It is concluded that WiFi is the best option for this application, as it is the least sensitive to interference relative to other common protocols. As basis for the WiFi implementation, an ESP32 is chosen because it has the wanted functionality and flexibility necessary for a prototype, and it was also conveniently in stock. The WiFi protocol was used with the ESP-NOW protocol on top of it, because ESP-NOW reduces the amount of complexity, supports encryption natively and allows out of band pairing. This protocol is upgraded with error detection and retransmission to achieve the reliability necessary. The out of band pairing is chosen to be implemented with a wired connection, facilitating I2C. I2C is chosen because of it's simplicity and use of acknowledgements. The ESP32 is programmed with ESP-IDF, which allows great freedom in programming the device.

With the use of WiFi, the device utilises a wireless communication protocol (**G1P**). By using encryption and out of band pairing, the communication channel is protected from eavesdropping (**G4P.a**) and is a safe connection ensured (**G4P.b**). Further research should however conclude whether this safe connection is of medical grade. The pairing is easy to confirm (**G4P.c**), with the use of a button and buzzer. Measurements have shown that the pairing works instantly for most of the time. This should of course never fail, so more research need to be done to fulfill requirement (**G5P.a**). The pairing allows any two transmitter and receiver to be paired with each other (**G5P.b**), this is however untested. The retransmission and error detection ensures that all signals are transferred lossless (**G6P.a**) up to a distance of at least 40 meters. Because of the way the transmission is implemented with buffers, the delay induced is exactly 0.5 seconds, which is the set by requirement **G6P.b**. The communication would be more reliable if this buffer size and thus delay would be increased.

Measurements showed that a data rate of 288kbit/s was achieved, which is 4 times more than necessary.

The communication module was delivered in time (**G10P**), which gave the ability to get a working prototype before the end of the project (**G10**). The prototype made by this subgroup is functional, but is yet to be integrated with the other subgroups.

8.2. Recommendations

There are several recommendations that can be made to further improve the quality of the product presented here. Firstly, and most importantly, there needs to be much more testing performed. Currently, the level of testing done is rudimental and limited, and certainly not adequate for medical equipment. Some tests that should be done are for example rigorous interference testing, pairing testing, battery life testing, temperature measurements of the chips, processor usage and testing the transmission in ambulance environments.

Another aspect that needs to be verified is code reliability. The product relies heavily on software, which is volatile and unreliable by nature. However, steps can be taken to verify that the written code is safe and reliable. More unit testing should be done, along with integration testing of the entire system. The ESP-IDF platform provides extensive testing facilities, which can be used to achieve the aforementioned goals. Additionally, LLVM's sanitizers [37] can be used to catch memory safety errors and undefined behaviour in the written code.

Because of the unreliability of the I2C pairing, it is reasonable to also explore different wire protocols for pairing.

References

[1] S.Speekenbrink K. Khalili. *Digital Signal Processing solution for a Wireless ECG*. Tech. Rep. Delft University of Technology, 2022.

[2] P.J. Wiersma and R.G. van Krieken. *Circuit Design for a Wireless ECG Device*. Tech. Rep. Delft University of Technology, 2022.

[3] Ambulancezorg nederland. *Ambulancezorg Nederland sectorkompas 2020, statistieken vanuit het RIVM*. 2022. URL: https://www.ambulancezorg.nl/sectorkompas (visited on 05/25/2022).

[4] Maria Sejersten et al. "Comparison of EASI-derived 12-lead electrocardiograms versus paramedic-acquired 12-lead electrocardiograms using Mason-Likar limb lead configuration in patients with chest pain". In: *Journal of Electrocardiology* 39.1 (2006), pp. 13–21. ISSN: 0022-0736. DOI: https://doi.org/10.1016/j.jelectrocard.2005.05.011. URL: https://www.sciencedirect.com/science/article/pii/S0022073605002219.

[5] M. Westendorp Dr. B. McGraw Dr. J. Lord. *Analysis and interpretation of the electrocardiogram, E-learning module, Queens University of Health Sci.* 2022. URL: https://elentra.healthsci.queensu.ca/assets/modules/ECG/normal_ecg.html (visited on 05/15/2022).

[6] AliveCor. *KardiaMobile 6L*. 2022. URL: https://www.kardia.com/kardiamobile6l/ (visited on 05/25/2022).

[7] Physio-control. *Lifepak 12 datasheet*. 2022. URL: https://www.physio-control.com/uploadedFiles/Physio85/Contents/Emergency_Medical_Care/Products/Operating_Instructions/LIFEPAK15_OperatingInstructions_3306222-002.pdf (visited on 05/26/2022).

[8] Koninklijke Philips N.V. *Philips Tempus ALS monitor*. 2022. URL: https://www.philips.nl/healthcare/product/HC989706000171/tempus-als-monitordefibrillator (visited on 05/30/2022).

[9] ZOLL, an Ashahi Kasei Company. *ZOLL Heart Failure Management System (HFMS)*. 2022. URL: https://cardiacdiagnostics.zoll.com/products/heart-failure-arrhythmia-management-system (visited on 05/30/2022).

[10] corpuls. *Corpuls3 Monitor*. 2022. URL: https://corpuls.world/en/products/corpuls3/#Monitoring-unit (visited on 05/30/2022).

[11] Oskar Flink. *Wireless electrocardiogram transmission based on ultra wideband radio*. 2018.

[12] Earl McCune. "DSSS vs. FHSS narrowband interference performance issues". In: *RF Signal Processing Magazine* (2000).

[13] "IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications". In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), pp. 1–3534. DOI: 10.1109/IEEESTD.2016.7786995.

[14] Espressif. *ESP-NOW User Guide*. 1st ed. July 2016.

[15] Sinem Coleri Ergen. "ZigBee/IEEE 802.15. 4 Summary". In: *UC Berkeley, September* 10.17 (2004), p. 11.

[16] Soo Young Shin et al. "Packet Error Rate Analysis of ZigBee Under WLAN and Bluetooth Interferences". In: *IEEE Transactions on Wireless Communications* 6.8 (2007), pp. 2825–2830. DOI: 10.1109/TWC.2007.06112.

[17] R Challoo et al. "An Overview and Assessment of Wireless Technologies and Co- existence of ZigBee, Bluetooth and Wi-Fi Devices". In: *Procedia Computer Science* 12 (2012), pp. 386–391. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2012.09.091. URL: https://www.sciencedirect.com/science/article/pii/S1877050912006825.

[18] Carles Gomez, Ilker Demirkol, and Josep Paradells. "Modeling the Maximum Throughput of Bluetooth Low Energy in an Error-Prone Link". In: *IEEE Communications Letters* 15.11 (2011), pp. 1187–1189. DOI: 10.1109/LCOMM.2011.092011.111314.

[19] Carles Gomez, Joaquim Oller, and Josep Paradells. "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology". In: *Sensors* 12.9 (2012), pp. 11734–11753. ISSN: 1424-8220. DOI: 10.3390/s120911734. URL: https://www.mdpi.com/1424-8220/12/9/11734.

[20] *ESP32 Datasheet*. 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (visited on 06/01/2022).

[21] *ESP32-WROOM-32e datasheet*. 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf (visited on 06/13/2022).

[22] P. van Mieghem. *Data Communications Networking*. 2nd ed. Cambridge University Press, 2011.

[23] Arun Kumar et al. "A comparative study of secure device pairing methods". In: *Pervasive and Mobile Computing* 5.6 (2009). PerCom 2009, pp. 734–749. ISSN: 1574-1192. DOI: https://doi.org/10.1016/j.pmcj.2009.07.008. URL: https://www.sciencedirect.com/science/article/pii/S1574119209000650.

[24] Kyuin Lee et al. "SYNCVIBE: Fast and Secure Device Pairing through Physical Vibration on Commodity Smartphones". In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018, pp. 234–241. DOI: 10.1109/ICCD.2018.00043.

[25] Frank Stajano and Ross Anderson. "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks". In: *Security Protocols*. Ed. by Bruce Christianson et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 172–182. ISBN: 978-3-540-45570-7.

[26] Jayant Mankar et al. "Review of I2C protocol". In: *International journal of research in advent technology* 2.1 (2014). ISSN: 2321-9637.

[27] Pamadi, Vishesh, and Bradford G Nickerson. "Getting Started With 1-Wire Bus Devices". In: *TR15-235, University of New Brunswick Fredericton, NB E3B 5A3 Canada August* 25 (2015).

[28] Dan Awtrey and Dallas Semiconductor. "The 1-wire weather station". In: *SENS(PETERBOROUGH, NH)* 15.6 (1998), p. 34.

[29] M Dilum R Perera, Ravinda GN Meegama, and MK Jayananda. "FPGA based single chip solution with 1-wire protocol for the design of smart sensor nodes". In: *Journal of Sensors* 2014 (2014).

[30] Ludmila Maceková. "1-wire-the technology for sensor networks". In: *Acta Electrotechnica et Informatica* 12.4 (2012), p. 52.

[31] Moi-Tin Chew, Tatt-Huong Tham, and Ye-Chow Kuang. "Electrical Power Monitoring System Using Thermochron Sensor and 1-Wire Communication Protocol". In: *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. 2008, pp. 549–554. DOI: 10.1109/DELTA.2008.92.

[32] N Montoya, L Giraldo, A Montoya, et al. "Remote monitoring and control system of physical variables of a greenhouse through a 1-wire network". In: *Advances in systems, computing sciences and software engineering*. Springer, 2006, pp. 291–296.

[33] *ESP32 API Documentation*. 2022. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html (visited on 06/03/2022).

[34] *Official IoT Development Framework*. 2022. URL: https://www.espressif.com/en/products/sdks/esp-idf (visited on 06/03/2022).

[35] *FreeRTOS reference manual*. 2022. URL: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf (visited on 06/03/2022).

[36] *FreeRTOS kernel documentation*. 2022. URL: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (visited on 06/03/2022).

[37] LLVM. *google/sanitizers*. URL: https://github.com/google/sanitizers.

A

Source Code ESP32

A.1. Pairing code transmitter

```
1  //I2C task used for pairing in the transmitter
2  static void i2c_task(void *priv){
3          struct pair_message msg = {0}, new_msg = {0}, to_send = {0};
4          memcpy(to_send.mac_addr, own_mac_addr, ESP_NOW_ETH_ALEN);
5          to_send.crc = esp_crc32_le(0, (uint8_t*) &to_send, PAIR_MESSAGE_DATA_SIZE);
6
7          while(1) {
8                  to_send.error = 0;
9                  vTaskDelay(200 / portTICK_PERIOD_MS);
10                 int size = i2c_slave_read_buffer(PAIR_I2C_SLAVE_NUM, (uint8_t*)
                    ↪  &new_msg, sizeof(struct pair_message),
11                                         1000 / portTICK_PERIOD_MS);
12                 if(size != sizeof(struct pair_message)) {
13                         continue;
14                 }
15
16                 // Skip message if CRC32 mismatches.
17                 uint32_t new_crc = esp_crc32_le(0, (uint8_t*) &new_msg,
                    ↪  PAIR_MESSAGE_DATA_SIZE);
18                 if (new_crc != new_msg.crc) {
19                         ESP_LOGD(TAG, "Skipping pair message because CRC32 doesn't
                            ↪  match! calc: %d, recv: %d",
20                                         new_crc, new_msg.crc);
21                         to_send.error = 1;
22                         i2c_slave_write_buffer(PAIR_I2C_SLAVE_NUM, (uint8_t*)
                            ↪  &to_send, sizeof(struct pair_message), 1000 /
                            ↪  portTICK_PERIOD_MS);
23                         i2c_reset_rx_fifo(PAIR_I2C_SLAVE_NUM);
24
25                         i2c_slave_read_buffer(PAIR_I2C_SLAVE_NUM, (uint8_t*)
                            ↪  &new_msg, sizeof(struct pair_message),
26                                         0);
27                         continue;
28                 }
29
30                 ESP_LOGD(TAG, "writing to i2c!");
31                 i2c_reset_tx_fifo(PAIR_I2C_SLAVE_NUM);
```

```
32                  i2c_slave_write_buffer(PAIR_I2C_SLAVE_NUM, (uint8_t*) &to_send,
                    ↪  sizeof(struct pair_message),
33                                  1000 / portTICK_PERIOD_MS );
34
35                  // Skip message if it's the same as the current one.
36                  if(memcmp(&msg, &new_msg, sizeof(struct pair_message)) == 0) {
37                          ESP_LOGD(TAG, "Skipping pair message because it's the same as
                            ↪  current message!");
38                          continue;
39                  }
40
41                  memcpy(&msg, &new_msg, sizeof(struct pair_message));
42                  ESP_LOGI(TAG, "Received new MAC address from master:");
43                  disp_buf(msg.mac_addr, ESP_NOW_ETH_ALEN);
44
45                  struct queue_event event = {0};
46                  event.type = WIECG_QUEUE_PEER;
47                  memcpy(event.peer_addr, msg.mac_addr, ESP_NOW_ETH_ALEN);
48                  memcpy(event.pmk, msg.pmk, ESP_NOW_KEY_LEN);
49                  memcpy(event.lmk, msg.lmk, ESP_NOW_KEY_LEN);
50
51                  if (xQueueSend(espnow_queue, &event, WIECG_QUEUE_DELAY) != pdTRUE)
52                          ESP_LOGE(TAG, "pair_i2c_slave: queue is full!");
53          }
54  }
```

A.2. Pairing code receiver

```
1   //I2C task used for pairing in the receiver
2   static void i2c_task(void *priv){
3           struct pair_message msg = {0}, new_msg = {0}, to_send = {0};
4
5           memcpy(to_send.mac_addr, own_mac_addr, ESP_NOW_ETH_ALEN);
6           // Fill key fields with initial random data.
7           esp_fill_random(to_send.pmk, ESP_NOW_KEY_LEN);
8           esp_fill_random(to_send.lmk, ESP_NOW_KEY_LEN);
9           to_send.crc = esp_crc32_le(0, (uint8_t*) &to_send, PAIR_MESSAGE_DATA_SIZE);
10
11          int slave_disconnected = 1;
12          while(1) {
13                  int ret;
14                  vTaskDelay(200/ portTICK_PERIOD_MS);
15                  if (!i2c_slave_exists(PAIR_I2C_MASTER_NUM, ESP_SLAVE_ADDR)) {
16                          slave_disconnected = 1;
17                          ESP_LOGD(TAG, "no slave connected!");
18                          continue;
19                  }
20
21                  if (!slave_disconnected)
22                          continue;
23
24                  slave_disconnected = 0;
25
26                  ESP_LOGD(TAG, "writing to i2c...");
27                  ret = i2c_master_write_slave(PAIR_I2C_MASTER_NUM, ESP_SLAVE_ADDR,
```

```
28                      &to_send, sizeof(struct pair_message));
29              if(ret != ESP_OK) {
30                      i2c_reset_tx_fifo(PAIR_I2C_MASTER_NUM);
31                      i2c_reset_rx_fifo(PAIR_I2C_MASTER_NUM);
32                      ESP_LOGE(TAG, "I2C write error:  %s \n",
                        ↪  esp_err_to_name(ret));
33                      continue;
34              }
35
36              vTaskDelay(100 / portTICK_PERIOD_MS);
37              ESP_LOGD(TAG, "reading from i2c...");
38              ret = i2c_master_read_slave(PAIR_I2C_MASTER_NUM, ESP_SLAVE_ADDR,
39                      &new_msg, sizeof(struct pair_message));
40              if(ret != ESP_OK){
41                      ESP_LOGE(TAG, "I2C read error:  %s \n",
                        ↪  esp_err_to_name(ret));
42                      continue;
43              }
44
45              if (new_msg.error) {
46                      ESP_LOGD(TAG, "There was an error in transmission! Flushing
                        ↪  buffer and trying again!");
47                      slave_disconnected = 1;
48                      i2c_reset_tx_fifo(PAIR_I2C_MASTER_NUM);
49                      continue;
50              }
51
52              // Skip message if it has an invalid CRC32.
53              uint32_t crc = esp_crc32_le(0, (uint8_t*) &new_msg,
                ↪  PAIR_MESSAGE_DATA_SIZE);
54              if (new_msg.crc != crc) {
55                      ESP_LOGD(TAG, "Skipping pair message because CRC32 doesn't
                        ↪  match! calc: %d, recv: %d",
56                                      crc, new_msg.crc);
57                      slave_disconnected = 1;
58                      continue;
59              }
60
61              // Skip message if it's the same as the current message.
62              if(memcmp(&msg, &new_msg, sizeof(struct pair_message)) == 0) {
63                      ESP_LOGD(TAG, "Skipping pair message because it's the same as
                        ↪  current message!");
64                      continue;
65              }
66
67              memcpy(&msg, &new_msg, sizeof(struct pair_message));
68              ESP_LOGI(TAG, "Received new MAC address from slave:");
69              disp_buf(msg.mac_addr, ESP_NOW_ETH_ALEN);
70
71              struct queue_event event = {0};
72              event.type = WIECG_QUEUE_PEER;
73              memcpy(event.peer_addr, msg.mac_addr, ESP_NOW_ETH_ALEN);
74              memcpy(event.pmk, to_send.pmk, ESP_NOW_KEY_LEN);
75              memcpy(event.lmk, to_send.lmk, ESP_NOW_KEY_LEN);
76
77              if (xQueueSend(espnow_queue, &event, WIECG_QUEUE_DELAY) != pdTRUE)
```

```
78                          ESP_LOGE(TAG, "pair_i2c_master: queue is full!");
79          }
80  }
```

A.3. Wifi task transmitter

```
1   //Wifi task transmitter
2   static void espnow_task(void *priv) {
3           ESP_LOGD(TAG, "espnow_task!");
4
5           int has_peer = 0;
6           esp_now_peer_info_t current_peer = {0};
7
8           struct queue_event event = {0};
9           struct message_queue_event message_event = {0};
10          int ready_to_send = 1;
11          int end_of_window = 0;
12          int started_sending = 0;
13          int tries = 0;
14          int retransmitting = 0;
15          while (1) {
16                  if (!has_peer){
17                          ESP_LOGI(TAG, "emptied queue");
18                          xQueueReset(message_queue);
19                          vTaskDelay(200 / portTICK_PERIOD_MS);
20                  }
21
22                  int delay = ready_to_send ? 0 : 100 / portTICK_PERIOD_MS;
23                  BaseType_t recvd = xQueueReceive(espnow_queue, &event, delay);
24                  if (recvd == pdFALSE)
25                          event.type = WIECG_QUEUE_NONE; // No espnow_queue event
26
27                  switch (event.type) {
28                          case WIECG_QUEUE_NONE:
29                                  break;
30                          case WIECG_QUEUE_RECEIVE: {
31                                  if (!has_peer) {
32                                          free(event.recv_data);
33                                          break;
34                                  }
35
36                                  struct wiecg_header *header = (struct wiecg_header *)
                                    ↪  event.recv_data;
37                                  if (header->type != TYPE_ACK) {
38                                          ESP_LOGW(TAG, "received packet other than
                                            ↪  ACK!");
39                                          free(header);
40                                  }
41
42                                  struct ack_message *ack = (struct ack_message *)
                                    ↪  header;
43                                  //ESP_LOGD(TAG, "received ACK for %d",
                                    ↪  ack->sequence);
44                                  if (retransmitting)
45                                          wiecg_bitmap_set(&tx_bitmap, ack->sequence);
```

```
46                              else
47                                      for (int i = 0; i < 11; i++)
48                                              wiecg_bitmap_set(&tx_bitmap,
                                                ↪  ack->sequence + i);
49                              free(header);
50                      } break;
51                      case WIECG_QUEUE_SEND: {
52                              if (!has_peer)
53                                      break;
54
55                              ready_to_send = true;
56                      } break;
57                      case WIECG_QUEUE_PEER: {
58                              if (has_peer)
59                                      ESP_ERROR_CHECK(esp_now_del_peer(current_peer.peer_addr));
60
61                              ESP_LOGD(TAG, "installing peer with: ");
62                              ESP_ERROR_CHECK(esp_now_set_pmk(event.pmk));
63                              disp_buf(event.pmk, ESP_NOW_KEY_LEN);
64
65                              current_peer.channel = 0;
66                              current_peer.ifidx = WIFI_IF_STA;
67                              current_peer.encrypt = true;
68                              memcpy(current_peer.lmk, event.lmk, ESP_NOW_KEY_LEN);
69                              memcpy(current_peer.peer_addr, event.peer_addr,
                                ↪  ESP_NOW_ETH_ALEN);
70                              ESP_ERROR_CHECK(esp_now_add_peer(&current_peer));
71                              disp_buf(event.lmk, ESP_NOW_KEY_LEN);
72                              has_peer = 1;
73
74                              // Signal that we're ready to send now.
75                              struct queue_event event = {0};
76                              event.type = WIECG_QUEUE_SEND;
77                              memcpy(event.peer_addr, current_peer.peer_addr,
                                ↪  ESP_NOW_ETH_ALEN);
78                              if (xQueueSend(espnow_queue, &event,
                                ↪  WIECG_QUEUE_DELAY) != pdTRUE)
79                                      ESP_LOGE(TAG, "espnow_recv_cb: queue is
                                        ↪  full!");
80                      } break;
81                      case WIECG_QUEUE_SEND_TIMEOUT: {
82                              // Timeout reached, check the bitmap if some packets
                                ↪  got lost
83                              // along the way.
84
85                              if (event.buffer_count < buffer_count)
86                                      break;
87
88                              int sent = 0;
89                              if (tries == 10) {
90                                      ESP_LOGE(TAG, "Retransmission timeout
                                        ↪  reached! Skipping to next window");
91                                      wiecg_bitmap_set_all(&tx_bitmap);
92                                      break;
93                              }
94                              retransmitting = 1;
```

```
95                              tries++;
96                              for (int i = 0; i < WIECG_BUFFER_SIZE; i++) {
97                                      // Skip packets that have already been ACK'd.
98                                      if (wiecg_bitmap_get(&tx_bitmap, i))
99                                              continue;
100
101                                     struct ecg_data data = {0};
102                                     if(wiecg_associative_array_read_specific(&tx_buf,
                                        ↪  i, &data))
103                                             continue;
104
105                                     sent = 1;
106                                     struct ecg_message ecg_msg = {0};
107                                     build_ecg_message(&ecg_msg, i, data.data);
108                                     ESP_LOGD(TAG, "retransmitting %d", i);
109                                     int ret =
                                        ↪  esp_now_send(current_peer.peer_addr,
                                        ↪  (uint8_t*) &ecg_msg, sizeof(struct
                                        ↪  ecg_message));
110                                     ready_to_send = false;
111                                     if (ret != ESP_OK) {
112                                             ESP_LOGE(TAG, "send error: %s",
                                                ↪  esp_err_to_name(ret));
113                                             break;
114                                     }
115                             }
116                             if (sent)
117                                     timer_start(TIMER_GROUP_1, TIMER_0);
118
119                     } break;
120                     default:
121                             ESP_LOGE(TAG, "Unknown queue event type of %d!",
                                ↪  event.type);
122                             break;
123             }
124
125             if(!ready_to_send)
126                     continue;
127
128             BaseType_t mes_recvd = xQueueReceive(message_queue, &message_event,
                ↪  0);
129             if (mes_recvd == pdTRUE) {
130                     uint8_t *message = (uint8_t*) &message_event;
131                     int ret = esp_now_send(current_peer.peer_addr, message + 1,
                        ↪  sizeof(struct message_queue_event)-1);
132                     ready_to_send = false;
133                     if (ret != ESP_OK)
134                             ESP_LOGE(TAG, "message queue send error: %s",
                                ↪  esp_err_to_name(ret));
135
136                     continue;
137             }
138
139             if (!has_peer)
140                     continue;
141
```

```
142                 // Now we can check the tx buffer.
143                 int count = 0;
144                 struct ecg_message ecg_msg[11] = {0};
145                 for (int i = 0; i < 11; i++) {
146                         struct ecg_data data = {0};
147                         uint32_t sequence = 0;
148                         int data_rdy = wiecg_associative_array_read_entry(&tx_buf,
                            ↪  &sequence,
149                                                 &data);
150                         if (data_rdy == -1) {
151                                 if (started_sending && !end_of_window) {
152                                         end_of_window = 1;
153                                         ESP_LOGD(TAG, "timer is turned on!");
154                                         tries = 0;
155                                         timer_start(TIMER_GROUP_1, TIMER_0);
156                                 }
157                                 vTaskDelay(10 / portTICK_PERIOD_MS);
158                                 break;
159                         }
160                         end_of_window = 0;
161
162                         //ESP_LOGD(TAG, "sending ecg data with sequence %d\n",
                            ↪  sequence);
163
164                         build_ecg_message(&ecg_msg[i], sequence, data.data);
165                         count++;
166                 }
167                 if (!count)
168                         continue;
169
170                 int ret = esp_now_send(current_peer.peer_addr, (uint8_t*) ecg_msg,
                    ↪  count * sizeof(struct ecg_message));
171                 ready_to_send = false;
172                 if (ret != ESP_OK)
173                         ESP_LOGE(TAG, "ecg send error: %s", esp_err_to_name(ret));
174                 retransmitting = 0;
175                 started_sending = 1;
176         }
177 }
```

A.4. WiFi task receiver

```
1  //Wifi task receiver
2  static void espnow_task(void *priv) {
3          struct queue_event event = {0};
4          uint32_t prev_seq = 0;
5
6          esp_now_peer_info_t current_peer = {0};
7          int has_peer = 0;
8
9          while (1) {
10                 BaseType_t recvd = xQueueReceive(espnow_queue, &event,
                    ↪  portMAX_DELAY);
11                 if (recvd == pdFALSE)
12                         continue; // Didn't receive anything, let's try again.
```

```
13
14                  switch (event.type) {
15                          case WIECG_QUEUE_RECEIVE: {
16                                  if (!has_peer) {
17                                          free(event.recv_data);
18                                          break;
19                                  }
20                                  struct wiecg_header *header = (struct wiecg_header *)
                                    ↪  event.recv_data;
21
22                                  uint32_t sequence =
                                    ↪  read_sequence(header->sequence_hi,
                                    ↪  header->sequence_lo);
23                                  prev_seq = sequence;
24
25                                  switch (header->type) {
26                                  case TYPE_ECG_MESSAGE: {
27                                          struct ecg_message *ecg_msg =  (struct
                                            ↪  ecg_message *) header;
28                                          struct ack_message resp = {0};
29
30                                          resp.sequence = sequence;
31                                          resp.header.type = TYPE_ACK;
32                                          int ret =
                                            ↪  esp_now_send(current_peer.peer_addr,
                                            ↪  (uint8_t*) &resp, sizeof(struct
                                            ↪  ack_message));
33                                          if (ret != ESP_OK)
34                                                  ESP_LOGE(TAG, "Error sending ack
                                                    ↪  response! %s",
                                                    ↪  esp_err_to_name(ret));
35                                          //ESP_LOGD(TAG, "Sent ACK for message %d",
                                            ↪  sequence);
36                                          testing_received_data();
37                                  } break;
38                                  case TYPE_BUTTON_MESSAGE: {
39                                          struct button_message *but_msg = (struct
                                            ↪  button_message *) header;
40                                          ESP_LOGI(TAG, "Received button data: %d",
                                            ↪  but_msg->pressed);
41                                          if (but_msg->pressed == BUTTON_PRESSED)
42                                                  gpio_set_level(BUZZER_PIN, true);
43                                          else
44                                                  gpio_set_level(BUZZER_PIN, false);
45                                  } break;
46                                  default:
47                                          ESP_LOGE(TAG, "Unknown message type of: %d",
                                            ↪  header->type);
48                                          break;
49                                  }
50
51                                  free(header);
52                          } break;
53                          case WIECG_QUEUE_SEND: {
54                                  //ESP_LOGI(TAG, "Send callback hit!");
55                          } break;
```

```
                        case WIECG_QUEUE_PEER: {
                                if (has_peer)
                                        ESP_ERROR_CHECK(esp_now_del_peer(current_peer.peer_addr));
                                ESP_LOGD(TAG, "installing peer with keys: ");
                                ESP_ERROR_CHECK(esp_now_set_pmk(event.pmk));
                                disp_buf(event.pmk, ESP_NOW_KEY_LEN);

                                current_peer.channel = 0;
                                current_peer.ifidx = WIFI_IF_AP;
                                current_peer.encrypt = true;
                                memcpy(current_peer.peer_addr, event.peer_addr, ESP_NOW_ETH_ALEN);
                                memcpy(current_peer.lmk, event.lmk, ESP_NOW_KEY_LEN);
                                ESP_ERROR_CHECK(esp_now_add_peer(&current_peer));
                                disp_buf(event.lmk, ESP_NOW_KEY_LEN);
                                has_peer = 1;
                        } break;
                        default:
                                ESP_LOGE(TAG, "Unknown queue event type of %d!", event.type);
                                break;
                }
        }
}
```

A.5. Bitmap API

```
struct bitmap {
        uint32_t *map;
        size_t size;
        size_t num_elems;
        SemaphoreHandle_t lock;
};

void wiecg_bitmap_init(struct bitmap *bitmap, size_t size);
void wiecg_bitmap_set(struct bitmap *bitmap, size_t index);
void wiecg_bitmap_set_all(struct bitmap *bitmap);
void wiecg_bitmap_unset(struct bitmap *bitmap, size_t index);
void wiecg_bitmap_clear(struct bitmap *bitmap);
int wiecg_bitmap_get(struct bitmap *bitmap, size_t index);
int wiecg_bitmap_check_all(struct bitmap *bitmap);
```

A.6. Re-transmission

```
for (int i = 0; i < WIECG_BUFFER_SIZE; i++) {
    // Skip packets that have already been ACK'd.
    if (wiecg_bitmap_get(&tx_bitmap, i))
        continue;

    struct ecg_data data = {0};
    if(wiecg_associative_array_read_specific(&tx_buf, i, &data))
        continue;
```

```
9
10      sent = 1;
11      struct ecg_message ecg_msg = {0};
12      build_ecg_message(&ecg_msg, i, data.data);
13      int ret = esp_now_send(current_peer.peer_addr, (uint8_t*) &ecg_msg, sizeof(struct
        ↪  ecg_message));
14      ready_to_send = false;
15      if (ret != ESP_OK)
16          ESP_LOGE(TAG, "send error: %s", esp_err_to_name(ret));
17  }
```