Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

Lenguajes y Sistemas Informáticos

Dispositivo IoT para la monitorización de pacientes de psiquiatría

Noriega Rodrigo, D. Iván

TUTORES:
TUTOR: Muñíz Sánchez, D. Rubén
COTUTOR: Díaz González, D. Juan

FECHA: Julio 2024

Table of contents

List of Figures

List of Tables

1. Abstract

This work extends the concept of a ”smart band” for mechanically restrained patients by Muñiz et al.. The system aims to maintain an unobtrusive monitoring solution while using common and cost-effective solutions, integrating hardware sensors, a microcontroller, software, and cloud computing. It focuses on timely alerts to medical staff for patient safety. The system’s data can later aid in critical incident detection and long-term analysis. It also offers remote configuration, adheres to safety regulations, and promotes openness and compatibility. This research presents a promising approach to enhance patient care in constrained settings.

2. Purpose and Scope

2.1.- Background

This builds upon the letter by Muñiz et al., in which they give hindsight into a possible implementation of a smart band for mechanically restrained patients [33].

2.2.- Idea and objective

The general purpose of this project is to build a system capable of monitoring mechanically constrained patients, while also being flexible, extensible and non-invasive for the end user. For this, a combination of hardware sensors, a microcontroller, software and cloud-served content will be used.

By building up on the background's idea of a "*smart band*", we create a minimally invasive framework for all the information we could need: heart rate, body humidity and temperature, movement, and noise. This could be refined further to add breathing detection in the future, but is not within the scope of this project due to it needing another, potentially tighter belt [42], which is difficult to communicate with the rest of the sensors and needs per-subject calibration and testing; or an external imaging system secured to the bed and pointed at the subject [23], which can have a negative mental connotation in addition to the communication problem.

It shall focus on the notification and alerts that could be given to the nursery staff, as stated in the background. This device will aid on the detection of dangerous incidents that could harm the patient, and notify another device which shall be carried by medical personnel, or be at a fixed place at their sight.

By giving this on-demand alerts, the patients can be treated in due time when needed, without constantly checking their vitals, which can be fatal in an under-staff condition; or simply due to bad timing, such as a cardiac arrest episode manifesting shortly after a nurse visit.

In addition, regular monitoring of the aforementioned variables can be used for later analysis in order to detect other problems by using techniques such as machine-learning (to automate sudden movement detection); or just by reviewing with a certified doctor, to discover erratic heart rate patterns or strange temperature variance.

The device itself should be manageable and configurable remotely, in case it needs maintenance operations such as re-calibration, firmware updates or improvements. It shall also comply with all electrical and safety regulations.

The system's software shall be open and free to explore and audit, and all dependant components of it shall be tested and accessible. Open and well-known standards should be followed to ensure compatibility and maintainability.

3. System Definition and Analysis of Alternatives

3.1.- Heart Rate Sensor

The core sensor of the system shall be the sensor responsible of heart rate measuring, as it is the primary factor to measure and act upon in an aggressive episode or other similar situations that are potentially dangerous to the patient under mechanical constraint.

As stated in the background, this will be an improved version of the draft proposed by Muñiz et al.. The usage of the AD8232 as a heart rate monitor front-end could be valid for developing a heart pulse analyzer resembling an electrocardiogram, as it offers more granularity and definition of all the pulse features by giving access to the analog signal that represents it.

The heart activity as measured by such an analog front-end is seen below (see Figure 3.1), with added labels for the various parts of the ECG wave that can be analyzed.

It can be broken down in segments and peaks:

1. PR interval,
2. PR segment,
3. QRS complex,
4. ST segment,
5. QT interval,
6. P wave,
7. Q wave,
8. R peak,
9. S wave,
10. T wave.

These parameters represent various properties of the heart pulses that can be interpreted and studied to discover possible heart problems [36], but that does not fall into the scope of this project. To measure the heart rate from this data, the wave shall be profiled in order to detect

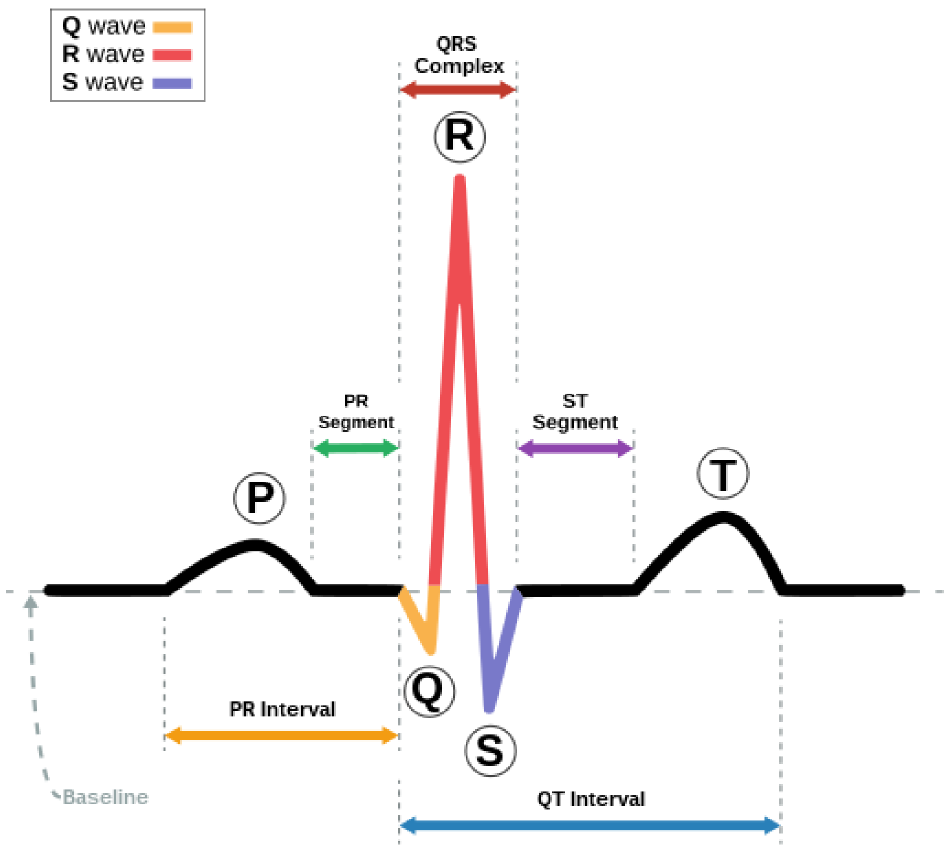

Figure 3.1: ECG of a heart in normal sinus rhythm. (by Atkielski)

the frequency of appearance of the QRS complex (which denotes the ventricular contraction), consequence of the heart pulse. This comes with drawbacks in case you just want to measure heart rate, as the signal has to be processed by a microcontroller using an algorithm to detect this frequency (also called R-R interval), thus deriving the pulses per minute.

By that logic, we can select a better suited front-end for our electrocardiogram subsystem that allows us to get the heart rate without constantly processing data. This is better in terms of **power usage** (as a dedicated chip can off-load the calculations from the main microcontroller), **data reliability** (pre-made algorithms tested in a controlled laboratory environment), and **ease of development** (no need to implement a QRS detection algorithm such as Pan-Tompkins [37])

The aforementioned chip is the MAX30003. This is a controller similar to the AD8232, as both are capable of reading an ECG from a single-lead configuration, which is perfect for our sports-band use-case; but the former is able to process the heart rate and R-R interval autonomously, with a very low power consumption (65 μW)[5]. It is generally available on different suppliers, and handles movement of the test subject without issues.

As the system is focused on heart rate monitoring, with other sensors complementing it, we have to adjust the hardware and software according to the chosen alternative for pulse detection. Given the selection of the MAX30003 chip due to its various advantages and availability with respect to other solutions, all other hardware should be fully compatible, and software should be developed such that the heart rate monitoring and data collection is not hindered in any way.

With it we can achieve our R-R detection by just importing a library and calling the needed functions. Using the MAX30003WING development board [30] to help with general system integration, the library of choice was `"max30003_protocentral"` by Protocentral, due to its feature completeness and it having been tested for our use-case. The supported connection by this library is SPI, or *Serial Peripheral Interface*, which can be found on pins *SCK*, *PICO*, *POCI* and *CS/SSEL*. This is explained further in section 3.4: Protocols and communication; subsection 3.4.1: SPI

3.2.- Microcontroller

It is clear that for the center of the system we need a microcontroller that can handle the signals generated and used by the MAX30003 chip. Also, the power, efficiency and memory of the chosen controller has to be taken into account. It shall also allow remote control and updating of the running program as stated in section 2.2. Given this premises, there are multiple alternatives to choose from:

3.2.1.- Raspberry Pi Pico

The Raspberry Pi Pico is a microcontroller board based on the Raspberry Pi RP2040 microcontroller chip [40]. It is a low cost device with 26 multi-function *GPIO (General Purpose Input Output)* pins, with multiple peripherals *(2 UART, 2 I2C, 2 SPI, 16 PWM channels)* and works with 3.3V logic, allowing its use with most commonly found sensors.

The board provides a Quad-SPI 2M flash for storing code and data. The *CPU* is a dual-core cortex M0+ running at up to 133MHz, with 264KB of *SRAM*, which is plenty for our use-case. It also has an *RTC*, sleep and low-power modes, a 12-bit ADC, and the main selling point of the chip: 2 Programmable IO (*PIO*) blocks, with 8 state machines in total.

This *PIO* blocks allow for custom assembly programming [41](p. 310) that operates at a very low level with a very fast speed, running on state machines outside of the main *CPU*, yielding the possibility of having additional peripherals that the system does not have built-in (including *DVI/VGA* video), having full control on a wave generation function, or integrating custom protocols for transmission over the wire.

Although the official guide for this device promotes the use of the Python (on its MicroPython implementation) programming language [22], the Raspberry Pi Foundation also provides a C/C++ *SDK* to build bare-metal applications in which performance might be critical. It also has a port of the popular Arduino core library for microcontrollers, *arduino-pico* [10], and allows remote *OTA* firmware updates for ease of management.

The Raspberry Pi Pico is a modern, well-rounded chip, but with the current amount of selected sensors, the powerful *PIO* capabilities would be left unused.

3.2.2.- ESP32

The ESP32 is a low cost series of microcontrollers made by Espressif. They feature built-in WiFi *(802.11b/g/n)* and Bluetooth *(v4.2 BR/EDR, BLE)* and a plethora of peripherals, including but not limited to: *12bit ADC, 2 8bit ADC, 10 touch sensors, 4 SPI, 2 I2S, 2 I2C, 3 UART, PWM* [48]. Different versions provide various features, such as in package PSRAM or flash memory, but they can also be implemented externally.

Depending on said version, it contains an Xtensa single or dual core 32bit *CPU* which can run at up to 240MHz; 448KB *ROM*, 520KB *RAM*, an *RTC* with 16KB of SRAM, and an *ULP*(Ultra Low Power) coprocessor to aid on low power peripheral monitoring, while using the main CPU sleep states to save energy. Power consumption in deep-sleep mode is as low as 10μA

The chip has 34 GPIOs (5 of them used for strapping, another 6 are input-only, and 6 more needed for flash memory) which work with 3.3V logic, like the Raspberry Pi Pico, making it suitable for most commercial hobby sensors. All output pins are able to generate PWM signals.

The module selected as the target for analysis was the ESP32-WROOM-32, due to its very high availability from a lot of distributors. It also provides an external Quad SPI flash, ranging from 4MB up to 16MB, and optionally, an external PSRAM. It has 2 versions; one with an integrated antenna, and another one with an external antenna connector compatible with *U.FL* from Hirose, *MHF I* from I-PEX, or *AMC* from Amphenol [12](p. 28). The pinout of the module is shown in Figure 3.3.

It supports a wide variety of programming frameworks and languages, ranging from MicroPython/CircuitPython, to pure C frameworks such as Arduino and ESP-IDF, or even less known ones like MongooseOS (supporting JavaScript)[3] or μlisp (supporting Lisp)[26]. It also allows remote management features such as *OTA* updates, and it is able to burn e-fuses to permanently save important, read-only information on the chip, such as MAC addresses or secure IDs. It is a solid and powerful chip which builds on top of the legacy NodeMCU (ESP8266) success, improving it in almost every way [13].

It is important to note that this was the microprocessor selected in the work from Muñiz et al..

3.2.3.- Arduino Family

The Arduino Family of microcontroller boards are very popular within hobbyists due to starting the revolution of makers with its low-cost open-source hardware, mixed with a simple and easy to use environment and coding framework. Created at the Interaction Design Institute Ivrea, in Italy, by joining together efforts from students and professors of an electronics class that wanted to create a board that they could use to teach electronics, while also being simpler, cheaper and more tailored towards their use-case [29].

The most commonly found amongst hobby projects is the Arduino UNO, which is a board based on the Atmel ATMega328P microcontroller [2]. It features 14 digital input/output pins (6 of which can be used as PWM outputs), 6 analog inputs, a 16MHz crystal oscillator, and the possibility of replacing the main microcontroller, to encourage users to push the hardware to its limits and not worry about burning the whole board. The microcontroller has 32KB of internal flash to hold data, a 1KB EEPROM, 2KB of internal SRAM, 2 8-bit and 1 16-bit timers, six PWM channels, an 8 channel 10-bit ADC, 1 SPI interface, 1 USART serial interface. It has a very low power consumption, with the power-down mode consuming as low as 1μA at 3 V.

Its main programming framework is the Arduino framework, which is well known within the maker community for its simplicity and ease of use, even when the underlying programming language is akin to C. Support outside of this framework is lacking. Connectivity has to be done with external modules through the SPI interface or UART, and remote management support is limited due to the very low power capabilities of the chip.

Although there are modern boards that are closer in power and capabilities as the previously mentioned ones, such as the Arduino MKR WiFi 1010 [1], we can see that the price is higher than the alternatives, and it still does not come close to the amount of peripherals and versatility the ESP32 or the Raspberry Pi Pico offer.

3.2.4.- STM32 Family

The STM32 family of microcontrollers are a bit less known on the hobby space. Nevertheless, they have been analyzed in order to investigate in some of the possibilities they can offer.

They have a wide variety of chips for a lot of different applications and use-cases [44]. As our primary concern is gathering data from sensors and sending them through WiFi or a similar alternative for further processing and analysis, the focus gravitates towards wireless-enabled chips from this family.

Although prices are similar to the powerful board alternatives listed above [45], they come with less GPIOs, flash memory and RAM, and they are a bare-bones chip. Shall you factor in the expenses of all the supporting parts and an external flash, and the small savings on the microcontroller alone are not worth the problems and design inconveniences that appear.

For example, if we take the STM32WBA52CEU6 (which at the time of writing costs less than $6) [46], it is cheaper than a whole ESP32-WROOM-32 development kit at €10 [32], or a Raspberry Pi Pico W at €8.40 [28]. But, given that we have more flash, a better development environment, and a pre-packaged kit for tinkering, as well as more GPIOs for future expansion, it makes sense to discard the STM32 as a valid alternative.

3.2.5.- Final microcontroller selection

Given all statements above, it is clear that the decision has to be taken between the Raspberry Pi Pico and the ESP32. They are very comparable and have similar features, performance, and price, barring exceptions such as the *PIO* from the Pico. That feature is not an advantage on this project as we are trying to follow standards, and use cross-platform and already implemented protocols for communication between the sensors and the microcontrollers. We decided to keep using the ESP32 in order to continue iterating on the ideas from Muñiz et al., as well as having double the storage for the program and other possible data, such as *SSL/TLS* certificates, logs or configuration.

3.3.- Other sensors

Complementing the microcontroller and the heart rate sensor are 3 more peripherals:

- **INMP441:** A microphone to obtain audio/noise data.
- **SHT21:**A temperature/humidity sensor.
- **MPU6050:** An IMU (*Inertial Measurement Unit*) to capture movement.

These devices give us more insights and parameters to take into account for study, and allow us more flexibility on alert definitions if necessary.

3.3.1.- INMP441 Microphone

A *MEMS* microphone that is cheap and readily available, with good digital audio and no big compromises for general usage. Allows the use of 24-bit *I2S*[25] to obtain the sound data digitally, with no analog conversions, which protects the audio quality from possible electromagnetic disruptions, as well as evading the possibility of the audio causing them to other sensors in the system.

An example of a non-important limitation of this microphone is its frequency response. With it ranging from 60 to 15.5kHz, we can capture virtually all of the important sounds a human can produce. The audio fidelity may be compromised if comparing it to commercial microphones (usually with a range of 20 to 20kHz), but for our use-case, we are not hindered by the loss of high frequency sounds.

As an omnidirectional microphone, it allows us to mount it in virtually any direction, without dampening the quality of the audio and the range of capture, greatly reducing the cost of product design. The only limitation by the high *SNR* of the device, so even though there is flexibility on the possible microphone orientations, looking for a placement close to the audio source is a must.

3.3.2.- SHT21 Temperature and humidity

A temperature and humidity sensor by Sensirion with very good performance/price ratio. This can allow monitoring body parameters if placed close to the test subject's skin. Communication is done using the *I2C* protocol, which will be explained in a later section (Subsection 3.4.2).

Containing a capacitive type humidity sensor, a band type temperature sensor, and a specialized analog and digital integrated circuit; the maximum resolution for this device is 0.04% RH and 0.01% ºC, with a typical accuracy of $\pm$2% RH and $\pm$0.3% ºC. Communication with the sensor is carried out through *I2C*, and can share a bus with other devices without an issue, allowing for a more efficient use of connections [43].

Some important details to take into account from the datasheet are the hysteresis and the response time. With them being 1% RH and 8 seconds respectively for the humidity, and between 5 and 30 seconds of response time for the temperature; we cannot expect a quick change on values, as a slow descent or ascent will show up in the data even if the environment suddenly changes.

As the aim of the device is to be in contact with body temperature and humidity, there shall not be any issues, a human body normally does not experiment sudden temperature or humidity changes. Nonetheless, there are no references to back this claim.

3.3.3.- MPU6050 Intertial Measurement Unit

A 6-axis (acceleration and gyroscope) *MEMS* motion sensor with an embedded low-power DMP (*digital motion processor*). Cheap and used in many other projects, it allows motion recognition in multiple ways, most importantly by itself with the aforementioned digital motion processor.

Communication is done through *I2C*, similar to the SHT21 sensor mentioned before, which allows for both of them to be on the same *I2C* bus, simplifying the PCB design.

With 6 degrees of freedom, 1KHz sampling rate and configurable sensitivity (4 steps for both accelerometer and gyroscope, from ± 2 to ± 16 g and from ± 250 to ± 2000 degrees per second respectively), it can be used for multiple scenarios, including general body motion sensing [24].

Distance calculation or space motion is discouraged, as without another frame of reference (such as a magnetometer), acceleration accuracy errors add up quickly and orientation changes can impact the end result. This could be leveraged with an external magnetometer connected to the second I2C port on the MPU6050, or replacing the component for another 9-axis IMU that includes a compass. This additional ”static” reference aids us in compensating for possible drift due to errors.

3.4.- Protocols and communication

Due to the very different nature of the used components and sensors, there are multiple communication protocols used in this system. Depending on the type of data they carry, the physical capabilities of the sensors, or the functional requirement they need to accomplish; a different protocol has been selected. Most of the times we are restrained to a specific one the closer we are to the actual sensor hardware, with more flexibility being gained as we go up in the abstraction layers (such as getting aggregated data out of the main microcontroller)

3.4.1.- SPI

The Serial-Peripheral Interface protocol is a full-duplex bi-directional serial communication protocol between multiple peripherals and a single controller. It requires a clock wire and two data wires for transfer in a bus, as well as an extra wire for each chip you add (and control) to the bus [8].

A theoretical example with a single peripheral and a single controller will be explained. For this connection, four wires are needed, and are commonly named as follows:

- **SCLK:** Serial Clock. This is a constant pulse on a specific frequency.
- **PICO:** Peripheral In, Controller Out. Data written from the controller into the external device.
- **POCI:** Peripheral Out, Controller In. Data read from the external device into the controller.
- **CS:** Chip Select. This signals the peripheral that the transmission is directed to it.

Older resources may refer to PICO as MOSI, POCI as MISO, or CS as SS. Those names have been deprecated by a large portion of hardware creators, and this work will continue to refer to them as the former [35].

In a scenario with more than one peripheral, only one extra wire per chip is needed, as that is the way for the protocol to address different devices connected in the bus. The other three wires can be shared.

Most of the disadvantages can be seen just by the wire requirements: Needing 3 lines for a bus, plus one extra line per device, can make board design hard and expensive when using a large amount of devices. This could be mitigated by *daisy-chaining* [8, Daisy-Chain method; Figure 7], but not all peripherals support it, and comes with the drawback of needing more clock pulses to transmit data, slowing communication.

3.4.2.- I2C

The Inter-Integrated Circuit protocol is a half-duplex bi-directional serial communication protocol between multiple peripherals and controllers. It only requires two signal wires to exchange information [49].

The base premise is having a general bus ”data” line, and a clock line with pull-up resistors in an open-drain configuration. All interactions are done by pulling the line low, which helps to alleviate possible problems when multiple devices try to send data at the same time. This fact, accompanied by the frame protocol definition, makes *I2C* a very robust and flexible due to not needing more than two signal lines. The only drawback is a more complex frame decoding phase and a half-duplex communication, which is fine in our case.

The following illustrates the basic communication flow on *I2C*:

$StartCondition- > AddressFrame- > DataFrame(s)- > StopCondition$

The *START* and *STOP* conditions operate the data and clock lines in a way not used on frames (as seen in Figure 3.5), for signalling. The frames of *I2C* consist of 8 bits of data and 1 bit for message acknowledge, which the peripheral has to assert at the correct time, or the controller will assume the peripheral did not receive the message (either there was a connection error or the device itself does not exist).

The peripheral can use *clock stretching* in order to respond the message acknowledge later, if needed. Clock stretching is an operation done by the peripheral device, which holds the clock line low for longer if needed in order to process a message. The controller will notice the lack of control and wait for the peripheral to release the clock line before continuing pulsing and sending data. This is the only instance in which the peripheral device is allowed control of the clock line [49, Section 6.2].

3.4.3.- I2S

The Inter-IC Sound (I^2S, or more commonly written *I2S*) is a single data line, unidirectional serial communication protocol between two devices. Designed for high data rates, it is focused towards (but not limited to) audio transmission. It requires a clock line (SCK), and a ”word select” (WS) line, which work as control lines for signaling the transmission of the data line (SD).

Unlike the other protocols, which used some type of addressing system, *I2S* focuses on speed and throughput, so there is no parameters defined on the protocol. The ”controller” device generates the constant clock signal, and switches the word select line from low to high to get data from the left and right channels, respectively. No clock cycle is wasted; one bit is transmitted at every clock cycle. The only norm is sending the MSB first. Therefore, the LSB of a word will always be preceded of the MSB of the next word, with no delays between clock cycles.

For the project's use case, the ESP32 acts as a receiver and a controller, signalling the INMP441 microphone the data rate, and the sound sample bits are directly placed into the data line.

On an ESP32, a *driver* is installed which simplifies the high data rate configuration and transmission, by the use of *DMA* or *Direct Memory Access*. This allows the microphone to "write" directly to defined memory buffers while the microcontroller is busy doing other tasks, and come back later to read and interpret the data as needed [11].

3.4.4.- MQTT

MQTT is a lightweight client/server publish/subscribe messaging telemetry protocol [34]. It is commonly used in IoT scenarios due to its simplicity, small code size and low network overhead. It runs over TCP/IP as it requires ordered, lossless bi-directional data transfer. It features a publish/subscribe pattern, in which the multiple clients can subscribe to different *topics*, and when a message is received by the broker, it will forward it to all the subscribers of that *topic*. It is agnostic to the payload content (so it can technically deliver binary data, even if it is unusual) and has a quality of service system for messages that can assure their delivery.

The ***topic*** is the key part of the system. It resembles an address, a place to publish a message; but in contrast to normal addresses, any client that has subscribed to the *topic* will receive the published message. The *broker* is the one in charge of message distribution over the subscribers, therefore it is usually named an *MQTT server*, and other connections are clients. This common naming is valid and can be easily understood in single-broker configurations, but the brokers can also be interconnected together, typically called a *bridge*.

The MQTT protocol is the choice of message transport protocol for this project, on a single broker configuration, due to its simplicity and ease of use on multiple devices. Standard measures and remote control will be implemented over this protocol, with out-of-band custom protocols developed for more time-optimized scenarios such as real-time monitoring.

3.5.- Alert handling

The alert system has to be, first and foremost, fast and reliable; all while being relatively secure, private, and easy to use. For this, a mixture of protocols and concepts will be used.

3.5.1.- FCM (Firebase Cloud Messaging)

As Android is the platform of choice for receiving alert notifications, the cloud platform that offers us the best developer and user experience is from the Android creators themselves (Google) with Firebase Cloud Messaging. The architecture of FCM is a simple pub-sub without polling, that can deliver messages to multiple types of devices. (see Figure [17])

Devices generate a *token* when installing your app, and that token subscribes to different *topics*, to which you can push notifications and data to be received quickly. There is no trouble of app background connections being terminated by the device, as this service is optimized and is known as *push*, meaning that the server itself ”pushes” the notifications to the device, instead of it asking (polling) the server at a fixed interval.

You can send messages directly to a specific device *token*, which allows for a more secure communication and more flexibility on notification options (with an important setting being the ”priority”), at the cost of having to send one request per end device.

In addition, there is a middle ground between *topics* and device tokens, called *device groups*. They offer the same benefits as device tokens, with the ability to group up to 20 devices [18]. The downside of this approach is having to manage the device group externally, using Firebase API, adding and deleting tokens as needed. This, according to the documentation, is typically oriented to a single user per client, which may have multiple devices. In the context of this project, they could be used as one device group per ESP32, but that functionality can be easily replicated as a mapping on the ”proxy” program between the MQTT alerts generated by the microcontroller and the Firebase API.

The message structure can be broken down into an address part (the *topic* or *token* that shall receive the message) and two information parts (*notification* and *data*). The address part and at least one of the information parts shall appear in the message, but mixed messages containing both types are possible. On Android devices, behaviour is different depending on which of the two information parts you send.

App state	Notification	Data	Both
Foreground	`onMessageReceived`	`onMessageReceived`	`onMessageReceived`
Background	System tray	`onMessageReceived`	Notification: system tray; Data: in extras of the intent

Table 3.1: Summary table regarding handling messages (from Firebase).

```
{
  "message": {
    "topic": "device-312",
    "notification": {
      "title":"(312) Device Alert!",
      "body":"Abnormal heart rate detected. Abnormal temperature detected."
    },
    "data": {
      "device_id": "312",
      "sensors": "hr,150,137,181;temp,38.7,38.2,39.1;",
    },
   "android": {
      "priority":"high"
    },
  }
}
```

Listing 1: Sample FCM message.

3.5.2.- MQTT and external alert forwarder

As the FCM authentication requires us to possess digital keys and secret credentials, we need to extract the requests into an external, secure service, that handles the alerts from the ESP32 and forwards them to FCM. This allows us to generate alerts without worrying about credentials being leaked or stolen if an ESP32 is analyzed, as well as apply additional checks before sending the actual notifications to mobile devices.

For this, a Python script was developed (referred from now on as ***alert forwarder***) that will run alongside the MQTT server, forwarding alerts from the IoT devices to the necessary end user devices. It is based on a quick-start code provided by Firebase [20] for the *connection token* and message generation, with custom logic added to connect and listen to the ESP32 alerts sent via MQTT.

3.5.3.- Android App

In order to receive the alerts generated by the ESP32 and forwarded to Firebase, a basic Android app installed on a device is needed. Its Firebase *token* shall be retrieved and used on the Python *alert forwarder* on the request to FCM.

As most of the notification functionality is provided by FCM, the development time of this app is minimal; nevertheless, the overall system can be improved by a lot just by iterating on this element. An easy-to-use management interface can be developed into the app (or a separate ”admin” app for safety) so that new devices can be linked. The user can get functionalities such as real-time monitoring on their phone, old sensor values and notification history.

As of now, all of this functionality is out of scope and will be documented later on the ”Future Work” section (4.5).

3.6.- Overall System

All final components of the system can be observed in the general overview diagram (see Figure 3.8), which highlights the main ”notifications” use-case with solid lines, and the supporting functionality with dotted lines. Physical support is not shown for every component, which will be analyzed in the following sub-sections.

An Internet connection is a must for the general system to work, or else, the alerting functionality using FCM would not work. Given a specific use-case of needing a more secure environment, with hardened restrictions, the Android app can be extended to connect directly to a locally hosted MQTT broker, so that it can listen to the alert *topics* of each device. This configuration is not in scope for this investigation and will not be analyzed; nevertheless, it is still possible to implement as explained above, taking advantage of the modularity of the system.

In the illustrated scenario diagram (Figure 3.8), all lines assume that the underlying hardware is connected and able to transmit information in some way, without limits. The system will not work if, for example, the Android device has no Internet connectivity to receive the notifications from FCM, or if the ESP32 has a poor or limited connection to the WiFi router.

The physical hardware needed to run the MQTT broker, the *alert forwarder*, and other supporting components, can be shared. This physical hardware is not illustrated in the scenario diagram as the system is agnostic to it, as long as the required functionality and networking is met. A WiFi router could provide connectivity for both the Android device and ESP32, and a single local server connected to it could host everything (except FCM); or the system could span multiple access points, networks and cloud machines with different paradigms.

3.6.1.- Physical support and hardware

As stated above, there are multiple ways to design the hardware requisites of the supporting software:

- *Cloud microservices and serverless*
- *Cloud VPS*
- *Local server*

Cloud microservices, or the microservice architecture, is a pattern on system design focused on division of tasks between multiple, dedicated small programs. The modularity of the architecture can help speed up development time, as well as build up resilience by the use of proxies and multiple nodes for a given part of the system. Lots of cloud providers offer tools to build microservice-oriented applications. All of this is inherently *serverless*, as you think about the purpose of the systems, and not the underlying hardware.

Serverless describes a recent paradigm based on the parallelization and sharing of physical servers, offering auto-scaling capabilities without the hassle of managing the underlying hardware. The users of this services can provide some code or machine configuration, and the system automatically scales the supporting servers in order to meet the demand. This allows them to have their application running without worrying about bottlenecks when the usage goes up.

The price of this paradigm depends on the amount of computer resources consumed over a given time. For example, imagine a system usually needs around 2Gb of RAM and uses 50% of a CPU core's compute power for scheduled tasks and normal user behaviour. But when office time starts, the system experiences a high load for over an hour, requiring 8Gb of memory and two CPU cores at 100% utilization. Adapting the program to a *serverless* paradigm allows it to automatically scale up the resources, and then scale them down when they are no longer needed. You are only billed by resources you used over a period of time. For the example above, 8Gb of RAM and two CPU cores are billed for an hour, and the 23 remaining hours only 2Gb and a single core is billed. This flexibility comes at a small extra, and a machine with a static machine with the baseline parameters is usually cheaper than the same configuration on a *serverless* system, even if there is no automatic scaling. Depending on the application, there are "hybrid" methods and architectures that mix standard machines for meeting the baseline demand, and a *serverless* system ready to handle peaks of activity.

For this project, the only sudden change of connection or information only occurs if multiple devices are connecting or disconnecting themselves for external factors, or a high volume of alerts are sent. A trial was executed with Amazon Web Services (AWS), using MQTT IoT Core for the MQTT broker, DynamoDB for data logging and Simple Notification Service (SNS) for alert listening and forwarding; it was discarded due to the high management overhead, and the free pricing tier could impose limitations for continuous use or higher frequencies [4].

Cloud VPS (Virtual Private Server) refers to a virtual machine with shared resources, rented out on a physical server that you do not control. It effectively works the same as a local physical server, with the benefits of being configurable to tailor your needs of memory and CPU power. The VPS is usually managed by the end user, although some providers offer managed alternatives. This approach can be useful for systems that need to scale every once in a while, or simple applications that only need a small amount of resources, and use them in bursts depending on the activity.

Providers also offer different kinds of servers to rent, with options ranging from a virtual machine with dedicated resources, full dedicated rental hardware, all the way up to co-location of your own hardware in their data centers. All of this is provider dependant.

For this project, the issue of data safety and security arises. Given that it is medical data, precautions shall be taken as to handle it correctly. Even if the parameters from the body logged by the sensors are inherently anonymous and only linked with the device that sends them (never with the person), it shall still be confidential. A cloud VPS or dedicated server could be useful for testing or even running the final product, but the usage of hardware not owned by a hospital (or hosted far from it) is discouraged due to the data concerns explained above. For this reason, the data acquisition has been done on either the development machines (acting as servers for the microcontroller needs), or with dedicated low-power hardware suitable for the supporting software.

The local server selection has to be able to handle multiple devices connected through the MQTT protocol, as well as saving their measures in a database. Any computing device more advanced than a microcontroller will suffice. A Raspberry Pi 4 has been selected for the proof of concept, as it showcases CPU, memory and networking capabilities powerful enough for the task, as well as enough room for possible future developments such as seizure prediction, all while being in a small form factor and very energy efficient (compared with classic x86 servers).

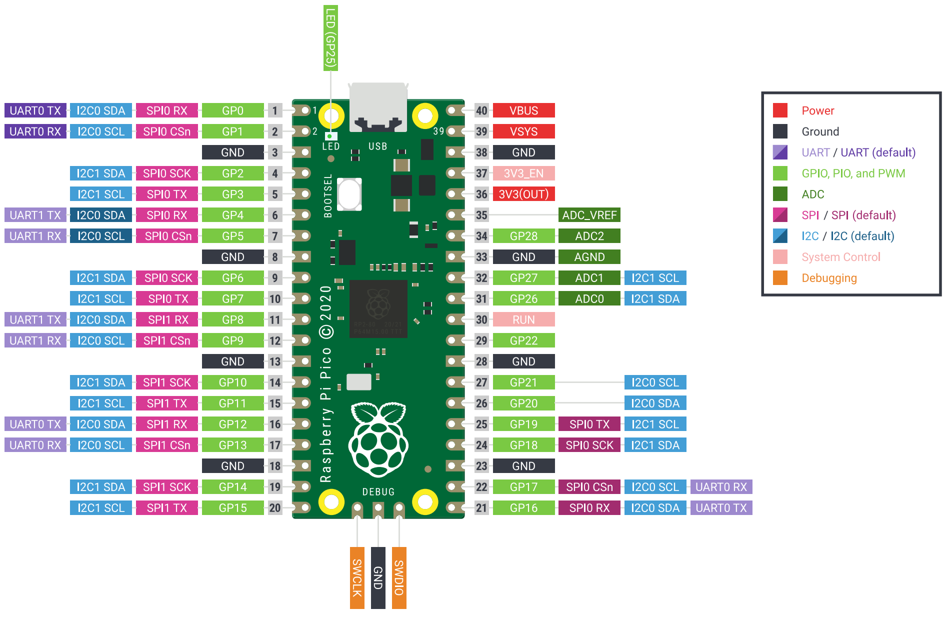

Figure 3.2: Raspberry Pi Pico pinout. [39]

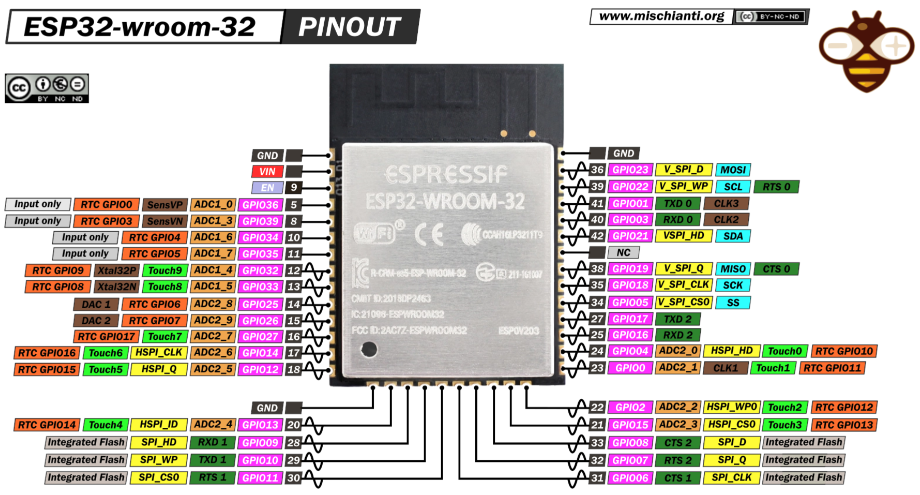

Figure 3.3: ESP32-WROOM-32 module pinout. [31]

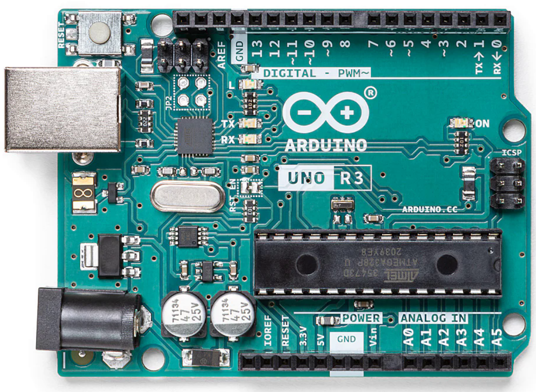

Figure 3.4: Arduino UNO Rev.3

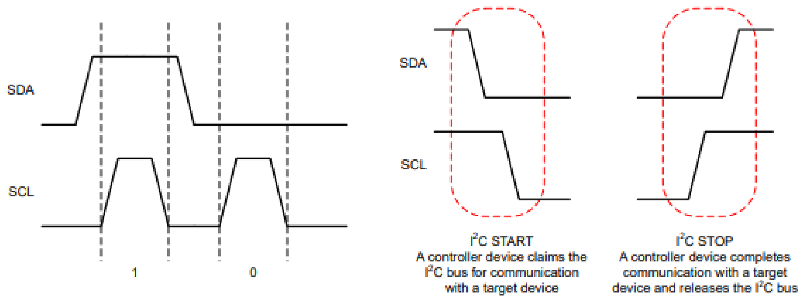
SDA
SCL
1
0
SDA
SCL
I²C START
A controller device claims the I²C bus for communication with a target device
I²C STOP
A controller device completes communication with a target device and releases the I²C bus

Figure 3.5: Digital bit representation and START/STOP signalling. (by Texas Instruments Inc., Fig. 3.2 and 3.1)

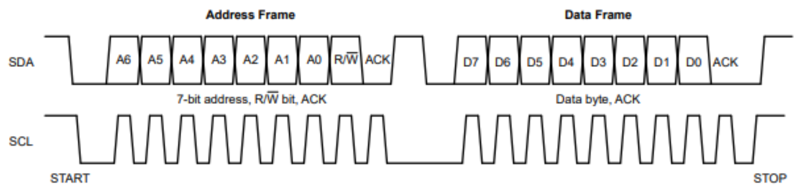

Figure 3.6: Frame of a simple I2C address and data transaction frame. (by Texas Instruments Inc., Fig. 3.3)

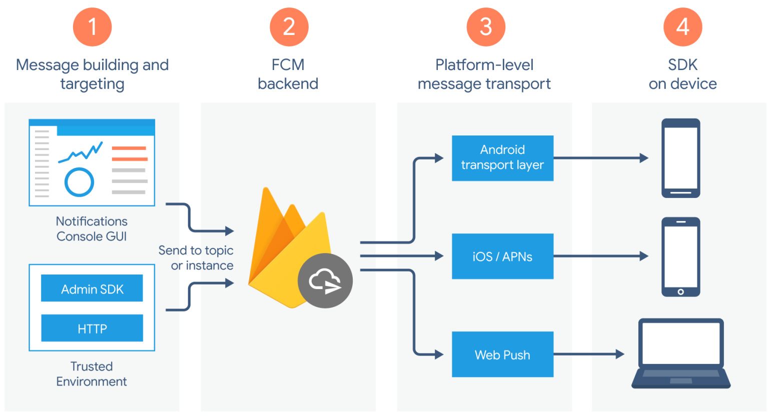
1
Message building and targeting
2
FCM backend
3
Platform-level message transport
4
SDK on device
Notifications Console GUI
Admin SDK
HTTP
Trusted Environment
Send to topic or instance
Android transport layer
iOS / APNs
Web Push

Figure 3.7: FCM Architectural overview.

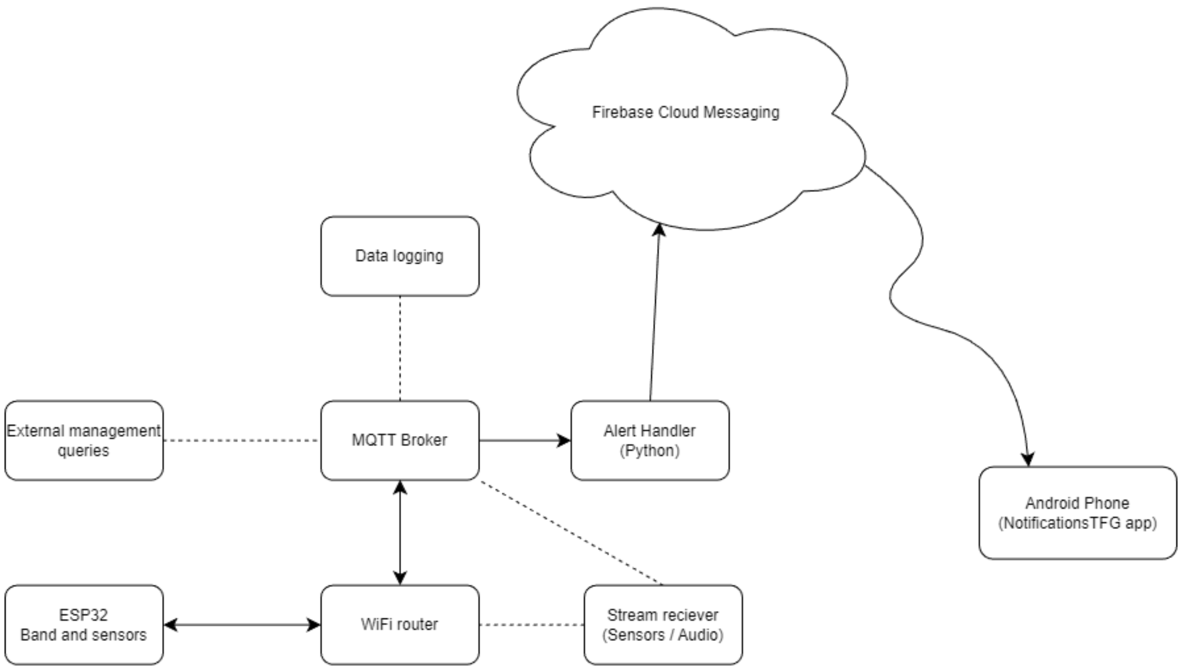

Figure 3.8: General overview of the system components and the data flow.

4. System Development

4.1.- Hardware Development

4.1.1.- First designs

For designing the first iterations of the hardware, a breadboard was used in addition to jumper cables in order to verify that the connection to the sensors was done correctly. This helped during the first design iterations and testing, as there was an agile way of switching connections in order to test different pin combinations. It was a crucial part on early development.

This method of coupling peripherals, while not recommended for long standing devices, is perfectly acceptable for testing a valid connection between the sensors and the micro-controller using external libraries. Focus at this stage was set on installing and using those libraries. High speed communication might not be available due to high resistances between the multiple wires, but should not be an issue for light data transmission.

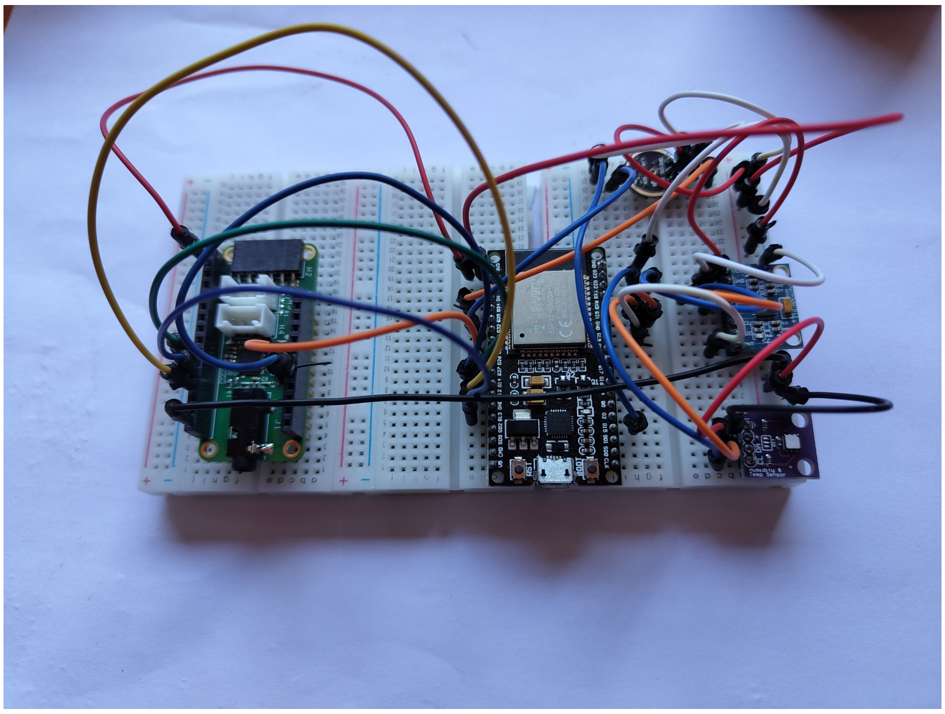
Figure 4.1: Final iteration of the breadboard connections.

Although there are recommended, default pins for each peripheral (see Figure 4.2), the ESP32 has an internal pin muxer and GPIO matrix [14] that allows us to use some peripherals (mainly lower-speed ones) on a wide variety of pins. In our use-case, we are only using one I2S, one shared I2C bus, and an SPI connection, so only the latter needs our attention. The pinout for our chosen board (generic ESP32-WROOM-32 devkit) is pictured below (see Figure 4.2).

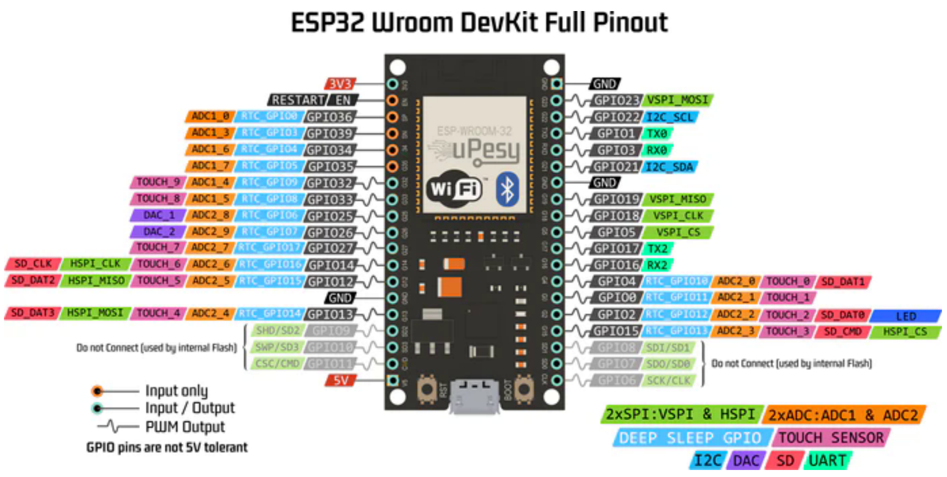
ESP32 Wroom DevKit Full Pinout
3V3
RESTART / EN
ADC1_0 / RTC_GPIO0 / GPIO36
ADC1_3 / RTC_GPIO3 / GPIO39
ADC1_6 / RTC_GPIO4 / GPIO34
ADC1_7 / RTC_GPIO5 / GPIO35
TOUCH_9 / ADC1_4 / RTC_GPIO9 / GPIO32
TOUCH_8 / ADC1_5 / RTC_GPIO8 / GPIO33
DAC_1 / ADC2_8 / RTC_GPIO6 / GPIO25
DAC_2 / ADC2_9 / RTC_GPIO7 / GPIO26
TOUCH_7 / ADC2_7 / RTC_GPIO17 / GPIO27
SD_CLK / HSPI_CLK / TOUCH_6 / ADC2_6 / RTC_GPIO16 / GPIO14
SD_DAT2 / HSPI_MISO / TOUCH_5 / ADC2_5 / RTC_GPIO15 / GPIO12
GND
SD_DAT3 / HSPI_MOSI / TOUCH_4 / ADC2_4 / RTC_GPIO14 / GPIO13
SHD/SD2 / GPIO9
SWP/SD3 / GPIO10
CSC/CMD / GPIO11
Do not Connect (used by internal Flash)
5V
GND
GPIO23 / VSPI_MOSI
GPIO22 / I2C_SCL
GPIO1 / TX0
GPIO3 / RX0
GPIO21 / I2C_SDA
GND
GPIO19 / VSPI_MISO
GPIO18 / VSPI_CLK
GPIO5 / VSPI_CS
GPIO17 / TX2
GPIO16 / RX2
GPIO4 / RTC_GPIO10 / ADC2_0 / TOUCH_0 / SD_DAT1
GPIO0 / RTC_GPIO11 / ADC2_1 / TOUCH_1
GPIO2 / RTC_GPIO12 / ADC2_2 / TOUCH_2 / SD_DAT0 / LED
GPIO15 / RTC_GPIO13 / ADC2_3 / TOUCH_3 / SD_CMD / HSPI_CS
GPIO8 / SDI/SD1
GPIO7 / SDO/SD0
GPIO6 / SCK/CLK
Do not Connect (used by internal Flash)
ESP-WROOM-32
uPesy
RST
BOOT
Input only
Input / Output
PWM Output
GPIO pins are not 5V tolerant
2xSPI:VSPI & HSPI
2xADC:ADC1 & ADC2
DEEP SLEEP GPIO
TOUCH SENSOR
I2C
DAC
SD
UART

Figure 4.2: ESP32-WROOM-32 module pinout [50].

4.1.2.- Sports-band for heart rate monitoring

A sports-band was used with a modified cable in order to have access of its electrodes through a 3.5 inch stereo jack (*TRS*), similar to audio ones, that connects into the MAX30003WING for heart monitoring. This cable was made for easy development, as the final product shall be encased in a cage that can clip onto this electrodes (like a general sports heart rate monitor). By soldering an end onto some button pins, they can be inserted into the sockets and get a 3.5 inch connector on the other side, while maintaining the possibility of using the band with other products. (see Figure 4.3)

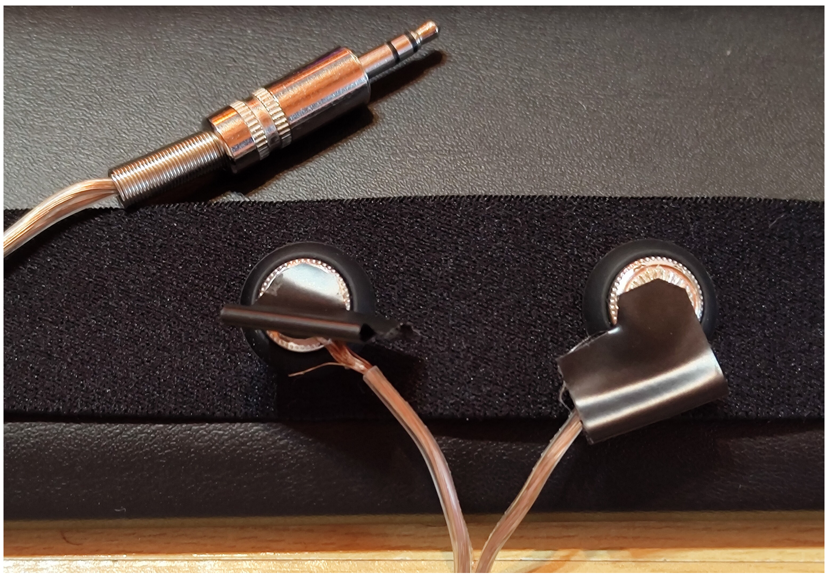

Figure 4.3: Connectors at the sports-band.

We have to keep in mind that having a long cable connected like this will increase the resistance (and thus the impedance) of the connection with the MAX30003 by almost 8% (see Figure 4.4), and it can even act as an antenna and capture interference from other devices around it, so our heart readings may not be fully reliable. Nonetheless, it is a good trade-off to make in order to get a more agile and comfortable development of the system, and it is the best way to run the device for debugging, as having it strapped to your chest could strain the *USB* connector used for it.

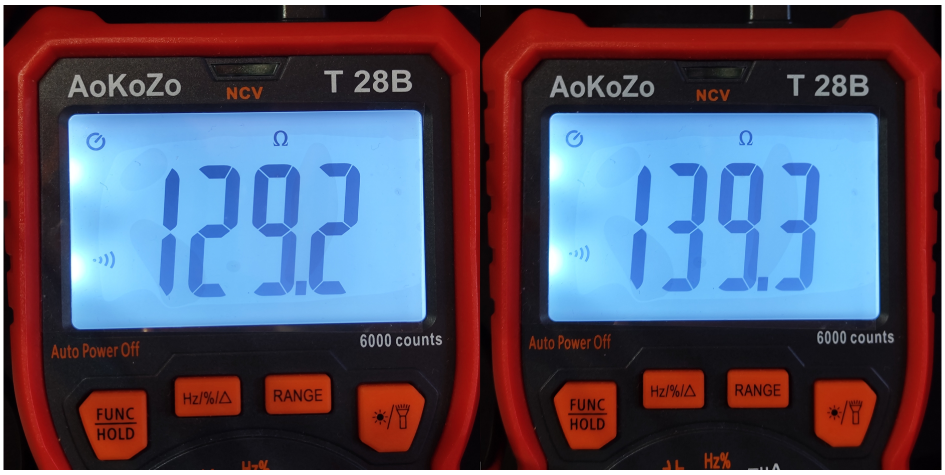

Figure 4.4: Left is band electrode to band socket, right is band electrode to 3.5 jack connector.

4.1.3.- Printed Circuit Board

After initial testing and program debugging, a diagram (Figure 4.5) was formalized in order to develop a basic prototype on a *PCB* (Printed Circuit Board). This ensures reliable connections between the components. The first prototype had some errors on production (2 missed traces, and a miscalculation of a board width), but as the affected pins were dedicated to low frequency of change signals like interrupts or chip selections, they could be fixed with some jumper wires (see Figure 4.6) in order to finish the development of the program, while having the high-speed traces in place for secure and safe data transmission.

4.1.4.- Second Circuit Board

To remedy the production errors of the first *PCB*, as well as to have a more compact design, a second circuit board was developed in a prototype board, soldering pin headers for the development boards of the sensors and the microcontroller, as well as the needed wires for connection. Due to the compact size, all the sensors are placed in one side, and the microcontroller lives in the other. This can make routing wires more difficult. Size was a general problem on this board, as some sensors were very close together and did not have enough room and pins around them to make ideal connections. Nevertheless, the diagram (Figure 4.5) was followed, getting a compact solution that could be encased in a box.

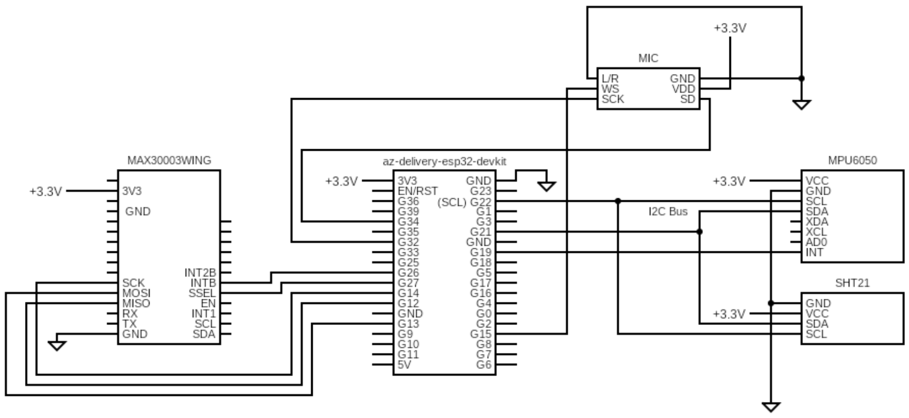

Figure 4.5: Diagram displaying the connections between the modules.

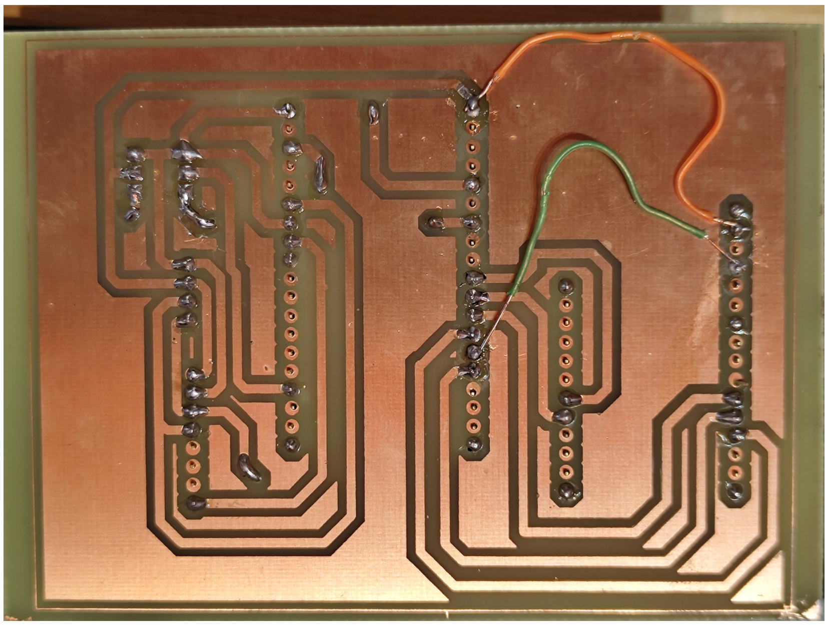

Figure 4.6: First iteration of the PCB (back side), with the patch cables.

On Figure 4.8, the connections between the sensor modules can be seen, and some rough patches can be appreciated. No short-circuits were detected, although the system needed external pull-up resistors for the I2C bus, or else some interference / corruption would appear on transmissions. The ESP32 was always reading an incorrect Z-axis acceleration from the IMU. It was tricky to debug, and the main hypothesis is that the data bus was floating too much time in an undetermined state, due to the internal ESP32 pull-up not being able to raise the voltage fast enough. It could be analyzed further with an oscilloscope, but it was not available during this development, and even then, it was quickly fixed by the addition of the external pull-up resistors, with a value of $4.7k\Omega$, recommended as a baseline for *I2C* connectivity (between $1k\Omega$ and $10k\Omega$, [49, Section 6.5]).

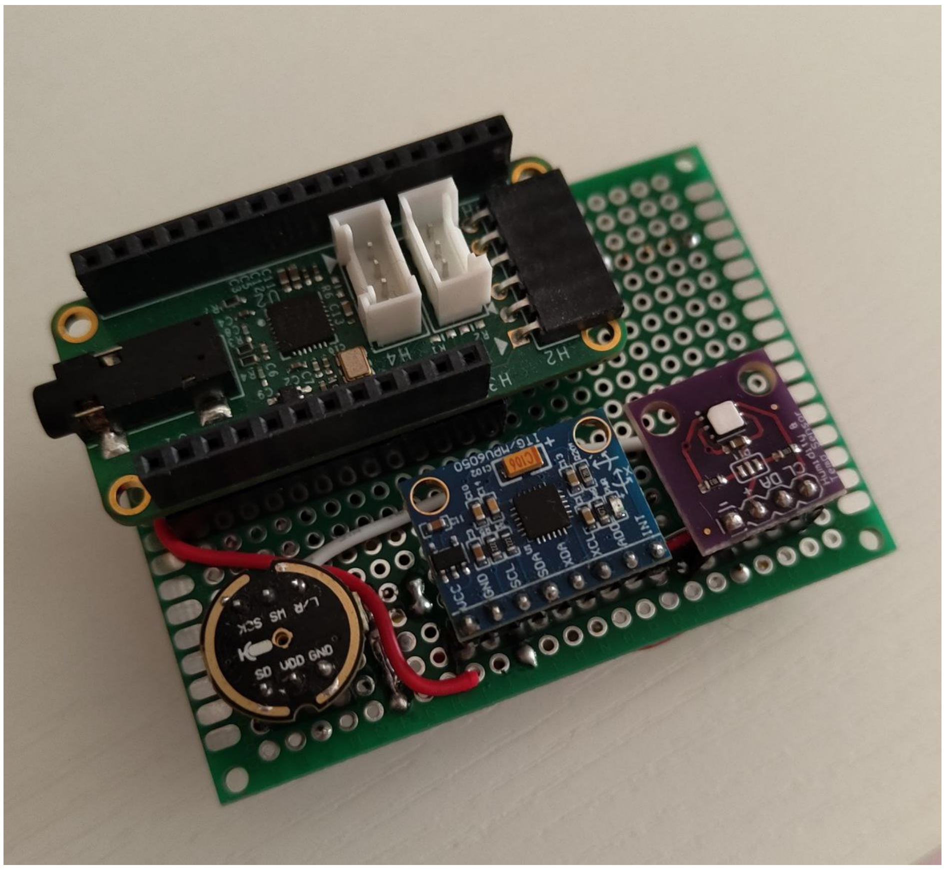

Figure 4.7: The second prototype board iteration.

4.2.- Microcontroller Software Development

The software was developed in the C++ language, on *Microsoft Visual Studio Code IDE* and using the plugin *PlatformIO*, which offers a great developer experience on a vast selection of microcontrollers. The frameworks chosen for the actual code were a mix of the *Arduino* framework (for access to a multitude of peripheral libraries), and the *ESP-IDF*, a framework specific for *Espressif* devices with useful code for inter-process communication, task handling and memory management.

4.2.1.- Module List

A module list is shown, detailing the functionality of every source file of the program loaded into the microcontroller.

File	Module	Description
main.cpp	General Setup	Initializes several device features such as Serial communication (for debugging), mounts SPIFFS and calls the configuration handler setup, initializes I2C and calls the sensor-specific initializations (as well as defning and registering the heart rate *ISR*), calls *wifimanager* to start the network connection, starts an *SNTP* client to get the current time, calls the MQTT setup function, and starts all the sensor tasks.
config.cpp	Configuration Handler	Loads the config.json located at the root of the SPIFFS filesystem on flash, into a global variable `configuration` for easy access anywhere on the code.
max3.cpp	Sensors: Heart Rate	Initializes the MAX30003 sensor and all required components to retrieve heart rate information. Responds to signals from the *ISR* to read the data from the sensor and load it into the needed structures and queues for later consumption.
imu.cpp	Sensors: IMU	Initializes the MPU6050 sensor and all required components to retrieve acceleration and gyroscope movement. Reads the sensor in an interval defined on `configuration` and loads the data into the needed structures and queues for later consumption.
sht.cpp	Sensors: Temperature and Humidity	Initializes the SHT21 sensor and all required components to retrieve temperature and humidity data. Reads the sensor in an interval defined on `configuration` and loads the data into the needed structures and queues for later consumption.

File	Module	Description
mic.cpp	Sensors: Sound	Initializes the INMP441 microphone, reserves memory for its *RingBuffer* and installs the I2S driver in order to load the audio data directly into the ESP32 memory. The I2S driver handles all of the reading according to the configuration we set statically upon compilation. A task reads the I2S data and writes it into the *RingBuffer* to allow the audio stream functionality, located on this module, but only enabled by an RPC call.
wifimanager.cpp	WiFi Connection Handler	Initializes the WiFi connection according to the access points set in `configuration`, and starts a task to maintain this connection or change access points if needed.
mqtt.cpp	MQTT Handler	Initializes the MQTT connection according to the `configuration` parameters, and starts three different tasks to enable message sending from anywhere in the program. A task runs the ”MQTT loop” which does all the queued networking; another task runs in an interval to send the maximum, minimum and average values of the sensors; and a final task handles the alerts generated by values outside of the thresholds defined in `configuration`.
rpc.cpp	RPC Functions	Upon initialization, a mapping is created that contains all the RPC functions: Methods that can be called remotely from MQTT.
stream_sensors.cpp	Real-time sensor data stream	Contains the FreeRTOS task, started via RPC over MQTT, that handles the data sensor streaming.

Table 4.1: ESP32 module list.

All modules (except *main.cpp*) have a companion header file with the *.h* extension, with the function definitions and variables needed for shared functionality with other modules, useful for code separation. Looped tasks are defined on their modules while being called and started from the main module. Additionally, there are three standalone header files:

- `credentials.h`: Hard-coded credentials for a secure MQTT connection, if defined in `configuration`.
- `stream_mic.h`: Defines only a single task (`void tMicTCPSender(void *pvParameters)`), which is called to start the microphone audio stream, with a *struct* formed of an IP address and port as a parameter. The function is defined in *mic.cpp* as to keep the audio functionality in the same module, and not have to share the *ring buffer* handle definition.
- `valuehandler.h`: Header containing all the relevant data containers. The handles for the sensor data queues, as well as their data *structs* and helper functions, are all defined here. Other generic *structs* such as the `params_ip_port` are found on this header, allowing multiple tasks and functions throughout the code to pass an IP address and port efficiently and in a structured manner.

In the following subsections, various modules will be explored, according to the program initialization sequence diagram found in the Annex 7.1. Built-in modules that implement functionality such as serial communication and logging (for debugging), WiFi, *OTA* update functionality or file system initialization are considered out of scope and not explored in this diagram.

4.2.2.- Library dependencies

Barring the frameworks themselves (Arduino and ESP-IDF), there is a number of libraries that the project depends on in order to work. On the built-in libraries side, the dependencies are:

- **SPIFFS:** The filesystem library chosen to store files on a predefined partition of the embedded flash.
- **Wire:** Native implementation for the I2C protocol.
- **SPI:** Native implementation for the SPI protocol.
- **WiFi / WiFiMulti:** Enable WiFi functionality, and support for multiple APs defined, to search for the best connection.
- **WiFiClient / WiFiClientSecure:** Network data connections via WiFi provided by the Arduino framework.

The following list shows the external libraries needed:

- **ArduinoJson *(bblanchon/ArduinoJson)*:** To aid in creating, serializing and deserializing JSON objects from/into native types.
- **MQTT *(256dpi/MQTT)*:** Manages the connection and messages between the ESP32 and the MQTT broker.
- **SHT2X *(robtillaart/SHT2x)*:** Interfaces with the SHT21 temperature and humidity sensor.
- **MPU6050 *(electroniccats/MPU6050)*:** Interfaces with the MPU6050 IMU.
- **Protocentral MAX30003 *(protocentral/ProtoCentral MAX30003 ECG AFE Sensor Library)*:** Interfaces with the MAX30003 ECG sensor.

The *Protocentral MAX30003* library has a small limitation: The chip select pin for the SPI communication with the module is hard-coded on a `#define` directive, and is not the same as the pin used in the project. The library was extracted and placed within the `lib/` folder of the project, with the directive modified to the correct pin, to ensure correct functionality. All the other libraries are defined as dependencies on the `platformio.ini` file, and obtained from the PlatformIO registry [38] during the build process.

4.2.3.- Configuration module

Being the manager of the configuration object, this module is the first one to be initialized. After the main module initializes the *SPIFFS* file system, the `loadConfigFromFlash()` function is called. Using the *ArduinoJson* library, the configuration file (`config.json`) is read from the file system and deserialized into a *JSON* object, available as a global object (`extern DynamicJsonDocument configuration`) for any module that imports the `config.h` header. No access restrictions are placed on the configuration; every module can read or write to it at any time.

This is a known anti-pattern [7] caught very late in development, when every module was already dependant on the external library object held in memory. The code is deeply coupled with the *accessors* and defaults that are offered. To remedy this, a C/C++ structure shall be created with the necessary parameters (with nested structures if needed) and custom logic should be implemented to load the available parameters into it, and set hard-coded defaults if needed. Then, all modules will use this structure directly, which removes the dependency.

This raises the question of thread-safety. Multiple reads at the same time are not an issue, but we must take care of a thread reading while another is writing, as the ESP32 is a dual-core device, so those threads can be executing concurrently. According to the ESP32 documentation for its FreeRTOS port, both cores have a ”symmetric” memory approach. If they both try to access memory at the same time, their calls will be serialized [15]. So our only worry is reading ”old” or stale data. No corruption will happen on whole *words* (up to 32-bit, as that is the architecture’s word size) even if both cores try to write at the same time. Nevertheless, if they try to do a sequence of read/add/write or similar, the final value will be the same as if only one of the cores were running.

So, in summary, all memory accesses are atomic on a *word* level, the only issues that may arise are partially written structures or missing operations on relative modifications. Absolute modifications (like the ones we are doing on the configuration module in section 4.2.11) are valid and safe.

4.2.4.- Value Handler module

This is a special module, as it does not have a source code file, it is a standalone header file. It contains the definition of multiple structures needed for data handling, as well as locks and queues. It also contains templated functions, in order to modify the values of a standardized structured called the `datum`.

```
/**
 * Basic structure for data aggregation.
 * Saves the avg/min/max and the count of the aggregated data points.
 * Initialize the datum by calling zeroDatum().
*/
template <typename T>
struct datum {
    uint32_t count;
    double_t avg;
    T min;
    T max;
};
```

Listing 2: The *datum* struct definition.

The modules will model their data structure using various `datum`, with a naming convention of (3-letter module name + *avg*). This structure generalizes the accesses to the properties of the periodically reported measures. In order to save as much data as possible while keeping a small size, a maximum, minimum and average value are continuously calculated by the `updateDatum()` functions. This functions are called by the different modules using *semaphores* as locks for access. This is the only instance of multi-core access lock used in the program, as we want the structure to be *atomic*, that being, the three parameters shall be originated from the same set of data at all times. If one core is zero-ing out the structure and another is writing to it, some parts of it may be zero and others have meaningful data, which renders the structure inconsistent. Serialized access to memory, as discussed before, will not help here due to the modification of multiple values.

The `checkDatumThreshold()` function template is called by each module when they update the different `datum` values on each sensor. The templated function is very useful, as different modules may use different `datum` variable types as long as they are numeric, but they all shall follow the same logic. If the values are over the configured thresholds, it notifies the `tMQTTAlert` task through the use of an *event group*, provided by *FreeRTOS*. Threshold checking through the parameter on `configuration` goes as follows:

- **Parameter *max*:** Notifies if the `max` value of the `datum` is higher than the parameter.
- **Parameter *high*:** Notifies if the `avg` value of the `datum` is higher than the parameter.
- **Parameter *low*:** Notifies if the `avg` value of the `datum` is lower than the parameter.
- **Parameter *min*:** Notifies if the `min` value of the `datum` is lower than the parameter.

Listing 3 shows the snippet of code that implements the `checkDatumThreshold()` function. The common anti-pattern talked about in subsection 4.2.3 of the external *ArduinoJson* library can be seen here, as it gets values from the global `configuration` object and uses them as the `JsonVariant` type from the library.

As a standard way of updating values, the modules send the new data on an interval defined on `configuration` through the queue (even if no real-time task is started), then acquire a lock using the semaphore and update the `datum` values of their specific structure using `updateDatum()`. After that, the lock is released and the threshold is checked using `checkDatumThreshold()`, with parameters according to the specified sensor and module. The MQTT alert task is notified if needed, using the `xEventGroup` series of functions given by FreeRTOS. The minimum, maximum and average values stored in the structures are cleared by calling `zeroDatum()` whenever a periodic report is sent. An example is explained below, on listing 4:

The `xEventGroup` series of functions allow us to ”notify” one or more tasks that are waiting on the object, like a shared semaphore, and carry a value that can be modified by the caller. It is used as a way to signal the MQTT task which module triggered the alert, without sharing task handles. The `valuehandler.h` module defines the *Event Group handle*, allowing any module to use it, but the MQTT alert task in the MQTT module is the one that initializes it.

4.2.5.- Heart rate module

In charge of initializing and reading the measures from the MAX30003 heart rate sensor, this module is crucial for the overall system. This follows the setup structure that other sensors do: Create the data structure for the minimum, maximum and average measures (`ECGavg`); the semaphore for locking them and the queue to communicate with the real-time stream task if it starts. But the sensor library itself is initialized on "R to R" mode. This will generate an

```
/**
 * Checks thresholds for a datum and notifies the alert task
 * with the provided bit if needed.
*/
template<typename T>
void checkDatumThreshold(datum<T> datum, const char* name, uint8_t bit) {
    // Thresholds
    JsonVariant max,high,low,min;
    bool notify = false;
    if(alert_events) {  // Check if the event group handle is valid
        max  = configuration["monitoring"]["thresholds"][name]["max"];
        high = configuration["monitoring"]["thresholds"][name]["high"];
        low  = configuration["monitoring"]["thresholds"][name]["low"];
        min  = configuration["monitoring"]["thresholds"][name]["min"];

        if(max.is<T>() && datum.max > max) notify=true;
        if(high.is<T>() && datum.avg > high) notify=true;
        if(low.is<T>() && datum.avg < low) notify=true;
        if(min.is<T>() && datum.min < min) notify=true;

        if(notify) xEventGroupSetBits(alert_events, bit);
    }
}
```

Listing 3: Threshold check implementation.

```
inline void _updateValues() {
    xQueueSend(ecg_queue, &(ecg->heartRate), 0);
    if(xSemaphoreTake(ecglock, 100)) {
        updateDatum(ecgavg->hr, ecg->heartRate);
        updateDatum(ecgavg->rr, ecg->RRinterval);
        xSemaphoreGive(ecglock);
        checkDatumThreshold(ecgavg->hr, "hr", BIT_HR);
    }
}
```

Listing 4: ECG datum update code.

interrupt on the sensor's *INTB* pin whenever an R peak is detected on the ECG (as shown in Section 3.1), and allows reading both the calculated heart rate as well as the interval of time passed between the new and old peak. There is no need to continually *poll* the sensor; reading data is done when the interrupt is received, giving us very low latency data acquisition.

This creates a task that differs from the other, time-based ones. To implement this interrupt detection functionality, the MAX30003 *INTB* pin has to be connected to an input-capable pin of the ESP32. In this case, it is connected to GPIO pin 26. The main module has the *ISR* definition for this interrupt, as well as registering that function to handle the change on the pin. The ECG data reading task is notified of the interrupt by the FreeRTOS ”task notification” features. Any task can send a signal to another task, given they have the task handle. As only the main module has access to the task handles, the *ISR* is defined in it, and sends the signal to the ECG task using the `xTaskNotifyFromISR()` function. The task, which is blocked on the `xTaskNotifyWait()` function, will resume execution when the *ISR* gives control back to the scheduler, and will read the values from the MAX30003 status register to obtain and save the heart rate and R-R interval data. This works similarly to a semaphore, or the aforementioned *Event Groups*, but it is only targeted to a single task, and readily available without any setup.

```
void tECGRead(void * param) {
    uint8_t statusReg[3];
    _readValues(statusReg); // first read to clear intr
    while(1) {
        xTaskNotifyWait(0,ULONG_MAX,nullptr,portMAX_DELAY);
        _readValues(statusReg);
    }
    vTaskDelete(nullptr);
}
```

Listing 5: ECG data read task that waits for it to be notified.

Interrupts should be as short as possible, as to not cripple other tasks or miss other important interrupts, therefore the data itself has to be read externally by a task. The following snippets of code show how the ECG data read task and the *ISR* are coded.

Executing FreeRTOS functions on the *ISR* itself can lead to errors due to scheduling issues, so some useful functions have a special version dedicated to run in this context. On a normal task, to notify another, you should use the `xTaskNotify()` function or derivatives. But here, you need to use `xTaskNotifyFromISR()`, which also accepts an additional `ctx_switch` argument. This argument will be set to true if the *ISR* has woken up a task with higher priority to the one that was interrupted, and should yield execution to it. The

```
void IRAM_ATTR isrECG() {
  BaseType_t ctx_switch;
  if(htECGRead != nullptr)
    xTaskNotifyFromISR(htECGRead,1,eSetBits,&ctx_switch);

  if(ctx_switch)
    portYIELD_FROM_ISR();
}
```

Listing 6: *ISR* that notifies the ECG task using its handle.

`portYIELD_FROM_ISR()` function signals the scheduler that it should continue normal execution, but first check for higher priority tasks that have woken up.

In this case, all sensor tasks are initialized with the same priority by the main module, so the last part of this code will never run. Nevertheless, it is an important piece of code to have, in case any future changes modify the priority of tasks.

This module uses the standard procedure of value updating: send via queue, lock `ECGavg` structure, update data, release the lock, check thresholds.

4.2.6.- IMU module

The standard initialization procedure is followed when the main module calls `IMUSetup()`: Create and empty the minimum/maximum/average data structures (`IMUavg`,`vIMUavg`), the locking semaphore and the queue for data streaming. After that, the MPU6050 library itself is initialized, the IMU accelerometer and gyroscope range is configured and they are calibrated if needed.

The main task for updating the IMU values does the standard procedure of value updating, with the added catch that it has two data structures: `IMUavg` and `vIMUavg`. The first one is the data coming directly from the sensor, but the second one (virtual IMU average) is made up of computed parameters, currently the acceleration magnitude. The computed parameters are not sent through the queue, as they are considered redundant.

The acceleration magnitude is calculated by obtaining the norm of the vector formed by the three acceleration axis, using the Formula (4.1) [33]

$$||\vec{a}|| = \sqrt{aX^2 + aY^2 + aZ^2} \tag{4.1}$$

For this, the values received from the MPU6050 have to be converted to *g-units* (g). The algorithm was optimized for precision and speed by minimizing divisions and floating point number usage, as the Xtensa LX6 CPU of the ESP32 has a limited *FPU* implementation. Single precision is supported by the hardware, but comes with caveats on FreeRTOS context switching, as it is assumed by default that tasks do not use floating point registers. Double precision floating point arithmetic is software emulated, which is more prone to errors, and consumes more CPU time. [15]

A *naive* approach would be to convert all three values to *g-units*, raise them to a power of two, and add them, before finally getting the square root. This will use multiple floating point operations, which we are trying to minimize The algorithm implemented tries to minimize floating point usage until it is absolutely necessary (*g-units* conversion and square root), operating on 32-bit integer numbers whenever possible.

```
double_t _vecMagnitude(int16_t ax, int16_t ay, int16_t az) {

    uint32_t x2 = (int32_t)ax * (int32_t)ax;
    uint32_t y2 = (int32_t)ay * (int32_t)ay;
    uint32_t z2 = (int32_t)az * (int32_t)az;

    return sqrt((x2+y2+z2))/ACCEL_MAX_RANGE_DIV;
}
```

Listing 7: Vector norm implementation for the raw MPU6050 values.

It may seem like too many operations are happening, but it can be optimized easily by the compiler to three integer multiplications (from the promoted 16-to-32 bit integers), two integer additions, and a final conversion to double precision floating point for the call to the built-in square root function and division. `ACCEL_MAX_RANGE_DIV` is a double precision floating point number containing the result of *32768* (the maximum absolute value of a signed 16-bit integer) divided by `ACCEL_MAX_G`, with `ACCEL_MAX_G` being the current range of precision of the accelerometer in *g-units*. The maximum value this variable will hold is 16384.0, as the minimum acceleration range of the MPU6050 is $\pm 2g$.

The base formula is derived on Equation 4.2, with R = `ACCEL_MAX_RANGE_DIV`, and lowercase ax, ay, az being the equivalent raw values from the MPU6050:

$$\begin{aligned}
\|\vec{a}\| &= \sqrt{aX^2 + aY^2 + aZ^2} \\
&= \sqrt{(\frac{ax}{R})^2 + (\frac{ay}{R})^2 + (\frac{az}{R})^2} \\
&= \sqrt{\frac{ax^2}{R^2} + \frac{ay^2}{R^2} + \frac{az^2}{R^2}} \\
&= \sqrt{\frac{ax^2}{R^2} + \frac{ay^2}{R^2} + \frac{az^2}{R^2}} \\
&= \sqrt{\frac{(ax^2 + ay^2 + az^2)}{R^2}} \\
&= \frac{\sqrt{(ax^2 + ay^2 + az^2)}}{R}
\end{aligned} \tag{4.2}$$

This value is finally used in conjunction with `checkDatumThreshold()` to send alerts if the amount of movement goes over the configured threshold, in *g-units*.

4.2.7.- SHT module

The standard initialization procedure is followed when the main module calls `SHTSetup()`: Create and empty the minimum/maximum/average data structures (`SHTavg`), the locking semaphore and the queue for data streaming. After that, the SHT21 library itself is initialized. As an implementation detail, the library expects initialization by calling the `begin()` method of the `SHT21` object, which will restart the sensor. It is advised to check for connection after this operation, but due to the speed of the ESP32, the setup flow needs to wait a bit before calling `isConnected()` on the `SHT21`, or else it will not respond correctly. This is easily done by the FreeRTOS function `vTaskDelay()`, which accepts the number of FreeRTOS *ticks* that it should sleep. In reality it returns control to the scheduler, but as we are still in the initialization phase of the program, no other tasks are running that could block or extend the sleep time. In order to convert between a time value (for example, milliseconds) and FreeRTOS *ticks*, the constant `portTICK_PERIOD_MS` is provided, signifying the period of a tick in milliseconds. Also, the convenience macro `pdMS_TO_TICKS()` is also provided, useful to make the code more readable. It is all used in conjunction on the following snippet to delay execution for 1 second (1000 milliseconds) between the call to `begin()` and `isConnected()`:

This module uses the standard procedure of value updating: send via queue, lock `SHTavg` structure, update data, release the lock, check thresholds. The thresholds are checked for both

```
void SHTSetup() {
    // **Earlier code ommited from the snippet**
    sht = new SHT21();
    bool ret_ok;
    ret_ok = sht->begin(&Wire);
    ESP_LOGD(TAG, "Begin returned %s", ret_ok ? "true" : "false");
    vTaskDelay(pdMS_TO_TICKS(1000));
    ret_ok = sht->isConnected();
    ESP_LOGD(TAG, "Connected returned %s", ret_ok ? "true" : "false");
    if((! ret_ok)) {
        ESP_LOGE(TAG, "SHT initialization error.");
        sleep(3);
        ESP.restart();
        while(1);
    }
    ESP_LOGI(TAG, "SHT ready.");
}
```

Listing 8: Delay on SHT initialization.

temperature and humidity. Due to the nature of the sensor, it is checked less often than the IMU, as the response time on the SHT (in respect to the real-life measures) is larger.

4.2.8.- Microphone module

The INMP441, as said before, uses *I2S* (explained in Subsection 3.4.3) to write data to the ESP32 memory. This module calls the required functions to automatically handle the data transfer, as the needed parameters are defined on its header as a constant. The standard initialization procedure is followed at the start, creating the `micavg` structure and the semaphore lock to access it, but there is no queue to send data to the real-time data stream. Instead, a ring buffer (circular buffer) is used to store the audio data as it comes, overwriting the old data if it is not consumed by the real-time stream. At this stage, it is only created by defining the needed size and calling `xRingbufferCreate()`, specifying that we want a byte buffer. This functionality is provided as a supplemental feature on the FreeRTOS implementation by ESP-IDF [16].

There are three different types of ring buffers according to the documentation:

- **No-Split buffers:** Intended to store items that must keep their structure contiguous in memory. Every item stored requires an additional 8 bytes for a header. Rounds item sizes to 32 bits.

- **Allow-Split buffers:** Intended to store items which structure does not need contiguous memory. Every item stored requires an additional 8 bytes for a header. Rounds item sizes to 32 bits.

- **Byte buffers:** Intended to store sequences of bytes, allowing reads and writes of arbitrary size within its bounds. Has no overhead, but does not guarantee 32-bit address alignment.

For this use case, the byte buffer is the perfect match, as we will be treating the audio data as a byte sequence. Memory alignment is not an issue, because this buffer is used to read the data from the internal *I2S* buffer and hold it until the real-time audio stream task sends it over the network.

After this buffer is initialized, a built-in *I2S* driver is installed according to the configuration set in the `mic.h` header. The project uses a standard configuration, with the necessary parameters adjusted in order to get a 22050 Hz sample rate, 16-bit single channel audio signal. There are 8 DMA buffers configured with 1024 samples each. At the aforementioned sample rate, each buffer will fill in 1024/22050 milliseconds, which comes in a total of approximately 371 milliseconds to fill the 8 buffers. The standard looping task that handles the microphone data has an execution interval of 100 milliseconds, so only around 3 buffers will be full at any given time, but they are reserved in case the scheduler cannot meet the delay demand and blocks the task for longer, so that audio is not lost.

The microphone data handler looping task simply executes on a similar flow to other modules (updating the `micavg` structure), except that it calculates every buffer until there is no more data left (while also inserting them into the ring buffer created prior), before calling `vTaskDelay()` for another 100 milliseconds.

A 22050 Hz sampling rate was chosen given the properties of the microphone (especially the frequency response) as well as the use-case (human noise detection). According to the Nyquist-Shannon theorem, perfect reconstruction of the analog audio wave is guaranteed given a sampling rate that doubles the maximum frequency cointained on the original wave. The microphone will register frequencies up to 11025 Hz on this configuration. Higher frequencies will be dampened by the low-pass filter included in the INMP441 itself [25], which helps with possible *aliasing*.

4.2.9.- WiFiManager module

Upon initialization, the module will load the WiFI access points saved in the `configuration`, check their validity, and add them to the `WiFiMulti` object offered by the external *WiFiMulti*

```
const i2s_config_t i2s_mic_config = {
    .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX),
    .sample_rate = 22050,
    .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
    .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT, // Ground the L/R pin
    .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_I2S),
    .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
    .dma_buf_count = 8, // 8 buffers, ~371 ms to get a full fill and lose data
    .dma_buf_len = 1024, // 1024/22050 = 46.44 ms to fill up a buffer.
    .use_apll = false,
    .tx_desc_auto_clear = false,
    .fixed_mclk = 0,
    .mclk_multiple = i2s_mclk_multiple_t(),
    .bits_per_chan = I2S_BITS_PER_CHAN_16BIT,
};
```

Listing 9: Configuration structure for the *I2S* driver.

library. Then, it offers the ”WiFi state” as returned by the `WiFiMulti.run()` call, in a global variable `WIFI_STATE` defined in `wifimanager.h`. This can help other modules determine the cause of the network error (no AP detected, general network errors...). It starts a small task that loops every 5 seconds calling `WiFiMulti.run()`, to update and maintain the WiFi connection state, as well as the global variable.

If the initialization procedure encounters an error connecting to the network, it will signal the device to restart. Errors during normal operation are allowed, and the library will handle reconnection gracefully thanks to the loop task started prior. A hard-coded single access point will be added to the `WiFiMulti` object in case the `wifi` parameter of the `configuration` is malformed. The following code snippet (Listing 10) shows the initialization procedure (without the loop task creation). Hard coupling and dependency with the *ArduinoJson* library can be seen here.

4.2.10.- MQTT module

The MQTT Setup starts after all the sensors are initialized and a WiFi connection is obtained. At first, the `MQTTClient` is created and configured according to the `"mqtt"` object in `configuration`. Then, the RPC system is initialized by calling `initRPC()`. Then, the MQTT connection to the configured broker is made, and a message handler (`hMQTTmsg()`) is registered to listen for RPC calls. A subscription request is made to the configured RPC *topic*. After that, the setup is considered complete, and three tasks related to MQTT are started.

```
uint8_t WiFiSetup(){
    WiFi.disconnect(true);
    WiFi.mode(WIFI_STA);
    if(!configuration.isNull()
      && configuration.containsKey("wifi")
      && configuration["wifi"].is<JsonArray>())
      {
        ESP_LOGI(TAG, "Using config WiFimulti...");
        JsonArray wifis = configuration["wifi"].as<JsonArray>();
        for(JsonVariant w : wifis) {
            if(w["ssid"].is<const char*>() && w["pass"].is<const char*>()) {
                ESP_LOGD(TAG, "Adding %s:%s\n",
                  w["ssid"].as<const char*>(),
                  w["pass"].as<const char*>()
                );
                wifiMulti.addAP(
                  w["ssid"].as<const char*>(),
                  w["pass"].as<const char*>()
                );
            }
            else{
                ESP_LOGW(TAG, "Skipping malformed wifi entry: %s",
                  w.as<String>().c_str()
                );
            }
        }
        WIFI_STATE = wifiMulti.run();
        _checkAndTryFixError(WIFI_STATE);
    }
    else {
        ESP_LOGI(TAG, "Using default WiFi parameters...");
        wifiMulti.addAP(WIFI_SSID, WIFI_PASS);
        WIFI_STATE = wifiMulti.run();
        _checkAndTryFixError(WIFI_STATE);
        //WiFi.begin(WIFI_SSID, WIFI_PASS);
    }
    // Start loop task...
}
```

Listing 10: WiFimanager setup procedure, verifying the WiFi parameters.

The **MQTT loop** (`tMQTTLoop`) is a bare-bones task that executes the `loop()` function of the `MQTTClient` object repeatedly, in one second intervals. This will manage the internal TCP connection to the broker, as well as send and receive any pending messages.

The **MQTT sender** (`tMQTTSend`) task is dedicated to the periodic reports of measures stored in the *avg* structures of each sensor. As the structures are locked by semaphores, the data is accessed in small chunks and stored in buffers inside the function, in order to lock the minimum possible time. Formatting and sending the message with the locks still acquired could result in delays or lost data in the sensor measuring tasks. In this minimal locking time, the `datum` inside the structures are cleared by calling `zeroDatum()`, so the *average* reported on this periodic messages is only calculated over that period of time. The period between this reports can be modified with the parameter `"update_period"` on the `"mqtt"` object inside `configuration`.

The **MQTT alert** (`tMQTTAlert`) task initializes the `alert_events` *event group*, used for notifying it of a current threshold alert. After an initialization delay to ensure the sensor values are settled, the task waits for any set bits on the *event group*, and analyzes the bits set in order to report a correct alert. The actual syntax of the message sent to *FCM* is done here, although it could be changed on-the-fly by the *alert forwarder*. The message is formatted using objects from the *ArduinoJson* library according to the *FCM* documentation, before finally being serialized and sent as an MQTT message.

As the MQTT loop task is running, any received messages on subscribed *topics* will be handled by the function defined on `onMessage()`, which is set to `hMQTTmsg` as described in the setup. This function will pass execution to the `processRPC()` function if it was sent to the RPC *topic*.

4.2.11.- RPC module

This module implements a simple but functional RPC system, compliant with the JSON-RPC protocol version 2.0 [27]. This version was chosen due to the comprehensive error messages it can return, without being too complex. RPC error messages are defined on the `rpc.h` header, on both *id* and text format, with specific names. The standard RPC errors are implemented, as well as some custom ones for specific functions, complying with Section 5.1 of the specification. The `data` object is always omitted.

All RPC functions are defined in this module, on `rpc.cpp`, and they do not appear on its header file as they should not be called from outside the RPC `processRPC()` function. If the functionality grows more, it could be distributed in different source files inside a folder, as long as the MQTT task is able to call the needed RPC function. In this case, to achieve that functionality, on the RPC module initialization, all functions are ”registered”: Their method name and a pointer to the function are added to a map structure. This structure was chosen as an easy way to get the function pointer knowing the method name. The `std::map` implementation offers a logarithmic time retrieval of the value of any key; this is better than any manual array iterative search implemented manually, and is provided as part of the standard library within the ESP-IDF framework. It can later be used to retrieve and call the needed function based on its registered method name. A preprocessor macro is used to simplify the task of updating the function map, passing a constant string as the method name and the function as arguments, which will be *casted* to the types that the map accepts.

The `processRPC()` function is the entrypoint to the RPC ecosystem, and it is the function that the MQTT task uses to handle RPC function requests. It handles message deserialization into a `JsonDocument`, returns errors if needed, finds the requested method and runs its function. When it finishes execution, it returns the response object back to the MQTT task, so it can send it to the broker.

All RPC functions have the same function signature/header: They accept a `JsonVariant` argument, which is a reference to the `"params"` object of the RPC message received over MQTT. They can modify the response by accessing the global variable `ret` (only under the RPC module scope), which is a `JsonDocument` object. Again, a hard coupling with the *ArduinoJson* library shows up, as well as another bad practice of having global variables. There are no race conditions or overwriting as the RPC processing is done synchronously by the MQTT module. No more than one RPC function may execute at the same time (although they may start tasks that run in the background outside its module, such as real-time data streaming)

Another preprocessor macro was created for the error processing, to aid on development and make the logic code more clear. Whenever an RPC function needs to signal an error back to the user (such as informing about wrong parameters, or internal errors) it should call `RPCError()` and pass the *id* of the error as defined on the `rpc.h` header. The macro makes use of preprocessor string concatenation to create the `"error"` object on the response message, with the `"code"` and `"message"` derived from the provided error.

A table (4.2) is shown detailing the RPC functions, with their method name, needed parameters and description:

Method	Parameters	Description
`rpc.list`	*None*	Returns a list containing all the RPC functions currently registered and available.
`config.get`	*None*	Returns the current `configuration` object.
`config.set`	*The new configuration, as a complete object.*	Updates the `configuration` to the provided parameter.

Method	Parameters	Description
`system.info`	*None*	Returns the microprocessor's uptime as microseconds, the current time, and the available memory heap size in bytes.
`system.reboot`	*None*	Restarts the ESP32. Will always return *OK* unless there is no memory to start the task that delays the restart.
`system.ota`	`"url"`: The complete URL of an HTTP server containing the binary firmware update.	Updates the microcontroller's firmware over-the-air, remotely. If successful, the device will restart.
`stream.sensors.start`	`"address"`: The IP address of the stream target. `"port"`: The target's listening port.	Starts the real-time sensor data stream, which will send data to the provided target. It will fail with `JRPC_ERR_RUNNING` if the task is already running.
`stream.sensors.stop`	*None*	Stops the currently running real-time sensor data stream task, if it is running. Will always return *OK*.
`stream.mic.start`	`"address"`: The IP address of the stream target. `"port"`: The target's listening port.	Starts the real-time audio data stream, which will send data to the provided target. It will fail with `JRPC_ERR_RUNNING` if the task is already running.
`stream.mic.stop`	*None*	Stops the currently running real-time audio data stream task, if it is running. Will always return *OK*.

Table 4.2: RPC functions.

Listing 14 shows a sample JSON-RPC message to start a real-time audio stream targeted at the IP address *192.168.4.8* and port *3334*:

```
// RPC Function type
typedef std::function<void(JsonVariant)> _rpcfun;

// RPC Function map
std::map<String,_rpcfun> functionMap;

/**
 * Helper function. It is a define macro because
 * we use the direct function, and then convert it
 * to std::function with the typename.
*/
#define registerRPC(name, func) \
  functionMap.insert({String(name), _rpcfun(func)})

/**
 * Initialize RPC function map.
*/
void initRPC() {
    // NOTE: add functions here to make them available.
    registerRPC("rpc.list", rpcList);
    registerRPC("config.get", getConfig);
    registerRPC("config.set", setConfig);
    registerRPC("system.info", systemInfo);
    registerRPC("system.reboot", systemReboot);
    registerRPC("system.ota", OTAupdate);
    registerRPC("stream.sensors.start", streamSensorData);
    registerRPC("stream.sensors.stop", stopStreamSensorData);
    registerRPC("stream.mic.start", streamMicData);
    registerRPC("stream.mic.stop", stopStreamMicData);

    for(auto const &pair: functionMap) {
        ESP_LOGI(TAG, "Registered function %s", pair.first.c_str());
    }
}
```

Listing 11: RPC function map setup.

```
#define RPCError(code) \
        JsonObject error = ret.createNestedObject("error"); \
        error["code"] = code; \
        error["message"] = code ## _TXT \
```

Listing 12: Preprocessor macro used for RPC errors.

```
void streamMicData(JsonVariant params) {
    if(params.isNull() || (!params.is<JsonObject>())) {
        RPCError(JRPC_ERR_INVALID_PARAMS);
    }

    if(
    (! params["address"].is<const char*>()) ||
    (! params["port"].is<unsigned short>())
    ) {
        RPCError(JRPC_ERR_INVALID_PARAMS);
    }
    // ** Function logic goes here... **
}
```

Listing 13: Sample usage of the RPC macro.

```
{
  "id":0,
  "jsonrpc": "2.0",
  "method": "stream.sensors.start",
  "params":{"address":"192.168.4.8","port":3334}
}
```

Listing 14: Sample JSON-RPC message.

4.2.12.- Real-time Sensor data stream module

This module only refers to the sensor data. The audio data is sent by a different task, by reading the ring buffer of the *mic* module explained in Subsection 4.2.8. Its implementation will be explained in the following subsection (4.2.13).

In order to get the measures out of the ESP32 and into the target device, outside of the MQTT broker, we need to establish a connection between both devices, and send data according to a specific protocol. The target of the protocol is to simplify the implementation on the microcontroller, including low overhead in serialization. For this, *TCP* was chosen as a base, and a simple byte structure was followed: a byte indicating the amount of ”structures” with data, followed by the actual objects. This is repeated 3 times, one for each type of sensor uploaded.

On the microcontroller, sensor measures are ”bundled” in queues, and emptied and serialized all at once at a fixed interval. This brings some benefits worth looking into:

- **More performance**: Bundling allows for less task switching and reduces the system calls for networking.

- **Packet optimization**: Multiple measures can fit in a single *TCP* packet instead of being divided in several ones, which may improve network performance at the cost of latency due to the bundling. This will depend on the networks *MTU*.

- **Different frequencies**: All three sensors could have different measuring frequencies and still work correctly. For example, the IMU may report values every 100ms, the SHT every 350ms, the ECG reports every time there is a pulse (usually between 700 and 1000ms) and you can send all the available information every 500ms.

The aforementioned serialization consists on reserving a big enough memory area and repeating the same operation for the three types of sensor (IMU, SHT and ECG): skip the first byte, dump the data from the queue in front of it, save the amount of structures dumped in the byte you skipped. For reading, the sequence is similar: Read the first byte and read forward that amount of objects, starting with the IMU, then SHT and finally the ECG. This allows for an arbitrary number of structures, including 0. The smallest possible packet is comprised of 3 bytes, all set to 0, indicating that there are no *IMUData*, *SHTData*, or *ECGData* structures to read on that packet. A simplified diagram is shown below (Figure 4.10).

While it is obvious that this protocol offers no forwards or backwards compatibility, no error correction, no timing information and no boundary safety checks; those negatives are justified by the benefits of simple multi-frequency compatibility, development speed, and small network overhead. This development is enough for the project scope and proof of concept.

The task that reads the audio data from the ring buffer and sends it through the network is started with an RPC call (`stream.sensors.start`) indicating the address and port, and stopped with its *stop* counterpart (`stream.sensors.stop`). A *TCP socket* will be initialized with the parameters provided. It will reset the queues as to not send old data, and start a loop of reading the queues, serializing the data and sending it through the sockets, every half a second. It will close the *socket* on any network error and try to restart this process indefinitely, until the aforementioned RPC *stop* call is made, which will kill this process and reap its stack memory.

The header just contains the task function definition, which is needed for the RPC module to create the actual task.

4.2.13.- Real-time audio stream

The real-time audio stream task is defined on the *mic.cpp* file itself, but has a specific header (`stream_mic.h`) in order to expose the task to other modules that require it (such as *rpc.cpp*).

The start flow is similar to the real-time sensor data stream task: Create a *socket* and clear the ring buffer before starting, but instead of having a loop on a defined interval, it waits for data on the ring buffer and sends it as soon as possible, on batches of 1024 bytes. This is done to minimize the latency on the audio stream, as the network overhead and the microphone data reading task already introduce a big delay. The ring buffer, when calling `xRingbufferReceiveUpTo()` to receive data, returns a pointer (and accepts a modifiable argument to return the size of the data contained), which can be both easily passed into the primitive `send()` function of the built-in *socket* library, and then freed from the ring buffer by using `vRingbufferReturnItem()` as to not overflow the internal ring pointers and potentially lose data.

```
void tMicTCPSender(void *pvParameters) {
    // **Earlier code ommited from the snippet**
    // Send data periodically
    while (true) {
        item = (char *)xRingbufferReceiveUpTo(
            ringbuffer_handle,
            &item_size,
            pdMS_TO_TICKS(1000),
            STREAM_BATCH_SIZE
        );
        if(item == NULL) {
          ESP_LOGE(TAG, "No data on ringbuffer for 1 second; continuing...");
          continue;
        }
        net_err = send(sock, item, item_size, 0);
        vRingbufferReturnItem(ringbuffer_handle, (void *)item);
        if (net_err < 0) {
            ESP_LOGE(TAG, "Data send error.");
            break;  // Exit the loop if sending data fails
        }
    }
    // **close the socket and try to reconnect**
}
```

Listing 15: Audio stream task receiving data from the ring buffer and sending it through a socket.

As the data is sent as a raw 16-bit samples over TCP, it can be reproduced using various methods, with a simple implementation explained in Subsection 4.4.4.

4.3.- Server Configuration

The Raspberry Pi was prepared with some external software to accommodate the system needs. Using docker, a container was created for the MQTT broker *mosquitto*. Another container (*telegraf*) was dedicated to store the periodic measures on a local database. As for the python supporting software, a *virtualenv* was created with the needed libraries, which will be used to run the *alert forwarder*.

Using *crontab*, the Firebase *alert forwarder* is set to run automatically at boot. The *mosquitto* and *telegraf* containers will also start up after the Raspberry Pi boots, thanks to the specified `restart` parameter on their `docker-compose.yml`.

4.3.1.- Alert forwarder and crontab

Cron is a Linux utility that executes commands at predetermined intervals or at a specific time, according to a user-dependant configuration file named *crontab*. It can be edited for the current user easily by running `crontab -e`. The file will open up on the default editor, with instructions on the basic syntax. The classic functionality is defining a specific time, and set "*" on fields that should not matter. For example, to run a script every hour at minute 0, the crontab entry will read like: `0 * * * * /path/to/command.sh`

The special `@reboot` parameter allows for running commands at system startup. No time is configured; the line should only contain the parameter and the command to run. This would be our complete example to run the Firebase *alert forwarder* startup script: `@reboot /home/pi/testBuzzer/supporting_python/firebase/start.sh`

The startup script will wait for a bit to allow the docker containers to initialize, and after that it starts the *alert forwarder* itself, on its *virtualenv*. It logs any console messages into a `latest.log` file, to facilitate tracing errors. This virtualenv is just a Python environment isolated from the system one, so that packages can co-exist without worrying about version mismatching.

```sh
#!/bin/sh

sleep 30
cd /home/pi/testBuzzer/supporting_python/firebase
.venv/bin/python -u main.py > latest.log &
```

Listing 16: Startup script for the *alert forwarder*.

4.3.2.- Docker configuration

Docker is an open-source platform allowing developers to build, deploy, run and easily manage containerized applications. [47] A container is a set of processes isolated from the host machine, similar to a *virtual machine*, but with less overhead due to the kernel being shared with the host.

Docker Compose is a tool for defining and running multi-container applications. [9] It offers a nice and developer-friendly way of defining what containers should run for a given system to work, including environment variables, shared networks, ports and files with the host system. On this use-case, it is used as a way to easily manage a single container.

The `docker-compose.yml` files follows the documentation to create a single container each (*mosquitto* and *telegraf*), with all the specific configuration files they might need mounted from the host filesystem into the containers. The network that connects the two is created externally, by executing `docker network create tfg`.

The directive `restart: unless-stopped` will guarantee execution of the containers until it receives a explicit stop command, this way, the *docker* engine will restart the containers when the host is booted up.

On the *mosquitto* side, the configuration allows any user to connect to the MQTT broker. It supports access control, users and passwords, which have not been configured for this proof of concept, and are reviewed on Subsection 4.5.6. It accepts connections on ports 1883 and 8883, with the latter being a TLS-secure option with a self-signed certificate, generated for this purpose.

As for *telegraf*, it is a server agent that collects metrics from various sources, transforms them and sends them to other outputs. It is used as a simple service to ingest the periodic measures sent over MQTT into a simple SQLite database, but it can be extended in the future to acquire more data and send the measures to different sources such as a PostgreSQL database.

4.4.- Supporting Software Development

Apart from the actual software embedded on the ESP32, and external programs and servers such as an MQTT broker implementation, there is some custom code needed for the full ecosystem to work.

```yaml
version: "3"
services:
  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mosquitto
    environment:
      - TZ=Europe/Madrid
    volumes:
      - ./certs:/mosquitto/certs
      - ./config:/mosquitto/config
      - ./data:/mosquitto/data
      - ./log:/mosquitto/log
    ports:
      - 1883:1883
      - 8883:8883
    restart: unless-stopped
    networks:
      - tfg


networks:
  tfg:
    external: true
```

Listing 17: Docker Compose YAML file for the mosquitto MQTT server.

4.4.1.- MQTT-Firebase alert forwarder

This is the main supporting component of the ecosystem. Being critical for alert delivery to end devices, this software should be running at all times alongside the MQTT broker, and have a reliable external connection to make the necessary requests to the Firebase API in order to ensure a correct delivery of notifications.

As explained in the overview of Section 3.5.2, it is programmed in Python, with external libraries needed for MQTT and Firebase communication. A Firebase example [20] was used as a base, iterating upon it with the MQTT client functionality needed for listening to the alert commands, and appending the Firebase device *tokens* needed for the FCM requests.

4.4.2.- Android app: NotificationsTFG

Already assessed on the overview of Section 3.5.3, a bare-bones Android App is necessary in order to receive fast and safe notifications. Using the *Android Studio IDE*, a "Hello world" blank app was created. Then, logic was added to ask the user for the notification permission, and to ensure the Firebase *token* is initialized and shown in the console.

The notifications are formatted directly from the ESP32 and modified by the *alert forwarder*, the App and FCM ecosystem display them directly on the notification bar of the device. No more development is needed to enable the phone to receive them. Additional features could be integrated in the future as explained in the overview, which will be expanded further on Section 4.5.

4.4.3.- Real-time Sensor data stream listener

For this functionality, a basic *TCP* server was developed in the Python programming language, in order to listen and unpack the sensor data sent as bytes directly from the ESP32, and display it on the screen.

After an standard *TCP* socket initialization listening on any address and a specified port (default 3334), the data processing loop receives all data sent by the microcontroller, and prints it to the screen, as well as plotting it into a 2D plot.

The built-in libraries `socket` and `struct` are used to create a *TCP* socket and unpacking the structures sent in the packets according to the protocol. Then, it is shown to the screen with a series of 2D plots using the `matplotlib` library. Multi-threading was used to handle the blocking nature of the network and the visualization parts. Currently, the CPython implementation of Python 3.11 is being used, which features the *GIL*. It is a CPython constraint that does not allow for multiple threads to access Python code and objects at the same time. Only one thread may have the lock acquired at any given point. The rest of the threads can only wait for it, or do computation and I/O outside of the Python realm. Therefore, the nature of this lock does not allow for true parallel execution under normal circumstances. As this is not a compute-heavy scenario, it is valid and functional for our purposes, as most of the time is spent on a sleeping state or waiting for data on the socket, which is where Python multi-threading model is best effective. The community is on the process of debating and implementing the option to turn off the *GIL* in future versions of Python [21].

4.4.4.- Audio data stream listener

A module for the ESP32 was programmed to send raw audio data from it into a *TCP* server. Creating a raw audio ingest server is not a trivial task, but receiving data through a socket and *piping* it through a well-established audio transcoder can be done with a few simple commands, given that we have the necessary programs installed.

To achieve this, a simple pipeline is launched, comprised of the *”ncat”* utility from *nmap* to open a socket, and *ffplay* (from *ffmpeg*) as the transcoder and audio player. Optionally, the built-in command *tee* (on *bash* or *powershell 7*) can be used in the middle, saving the audio stream to a file while still playing it in real time. A sample command is shown below:

```
ncat -l -p 3333 | ffplay -f s16le -ar 22050 -ac 1 -
```

This will launch *ncat* and make it create a listening TCP socket on port 3333, and ”pipe” the data received into the standard input of the next program; which is *ffplay*, configured to play a raw signed 16-bit (little endian) audio signal, with an audio frequency of 22050, and a single (mono) channel, from its standard input.

With this, we have a simple setup to listen to the audio picked up by the microphone in almost real time. Tests on a local network (ESP32 wireless ->WiFi router wired ->switch wired ->PC) show a latency of around 5 seconds. Network usage shall average 352.8 kbps, as that is the actual bitrate of the raw audio stream at this configuration (22050 Hz, 16-bit mono); nonetheless, in real tests, actual network usage will be higher due to *TCP* and lower *OSI* layer headers, and bandwidth consumed will fluctuate depending on buffer sizes, flush timing, and general network conditions.

This functionality can be enabled on-demand with an RPC call to the ESP32 specifying the server and port to connect to, but some interesting improvements can be developed in the form of a recording framework that, in case of a predefined sensor threshold or scenario, automatically started an audio stream session into a server, saving the data for later analysis.

4.5.- Future work

As the project is limited to a proof-of-concept system with basic features, there is multiple functionality that can be useful but has not been implemented. This section will overview some ideas as well as pending work to facilitate the use of this system in the future.

4.5.1.- Sports band compatible board

The current board iterations do not have in mind the wearable that they should be attached to. A production board should include all the current hardware without the development boards, as well as a battery and circuits needed for managing it, or a external power supply compliant with medical safety protocols. This board should attach effortlessly to the sports band on its front connectors, and have body contact for the temperature and humidity sensor.

A *PCB* should be designed following this needs of compactness and safety, as well as an enclosure to protect both the board and the patient.

4.5.2.- Android app improvements

The bare-bones app for the implementation of notifications is sufficient for its purpose; nevertheless, multiple features come to mind when thinking of a phone app for an IoT monitoring system:

- **Device management:** A way to manage new devices and assign them to be viewable or not on the app would be helpful for end users. Any phone that ”registers” a device shall receive its notifications, so the *alert forwarder* has to see this information.

- **Measure visualization:** Plotting and exploring historical periodic measures, while not trivial, is a feature given for granted in any measure-obtaining system. Some kind of connection is needed with the database, and a plot library shall be implemented on the app.

- **RPC function execution:** Ranging from basic information view to reconfiguring the `configuration` parameters, having a way to easily change the ESP32 behaviour from the phone is a very good quality of life feature for the end users.

- **Real-time measure visualization:** Allow the app to receive the direct measures from the ESP32 with the press of a button, and plot them on screen.

4.5.3.- Reactive data logging

The ESP32 still has some memory that is unused. A good use of this memory could be a ”replay” of sorts to be sent when any alert is triggered. Imagine the situation of a threshold that is commonly reached for a patient, but the periodic measures do not offer any insights as to why that happens. A system could be developed so that the ESP32 is continuously recording the real-time data into a ring buffer of a given size, and is uploaded when an alert is triggered. This way, the last few minutes of real-time data before the threshold is reached are available for analysis. The implementation could be similar to the real-time audio send, with a built-in ring buffer, except that it will always be overwriting itself until an alert is fired. At that point, the ESP32 can check its `configuration` to send the whole ring buffer data into a specified server, which will be listening and saving it into a database or a file for later use.

4.5.4.- Real-time stream improvements

The overall implementation of the real-time data streaming is valid. But even over a local network in an specific binary format, transmitting data over an insecure TCP connection brings some concerns. An effort should be made in the future to convert this plain TCP connections into secure ones, using protocols like TLS, allowing for confidentiality of the transferred data. This will cause some overhead in data transmission due to the data conversion, but it is a good trade-off for safe use of the real-time feature on less secure networks. Both the ESP32 code and the server implementation have to be compatible with the change. Also, the current ESP32 implementation only allows one stream type at the same time (so a maximum of one sensor stream and one audio stream). An interesting feature could be expanding this to multiple receivers, be it with the connections done directly from the ESP32, or through a distribution server.

Furthermore, a `configuration` parameter could define whether to start the real-time streams on device startup, as a way to have a permanent data ingestion; or another parameter to start a real-time stream whenever an alert occurs. The latter, paired with the reactive data logging defined before (Subsection 4.5.3) could be a very competent mix, achieving low, aggregated data transmission on normal situations, but adapting to a fast data transmission on important scenarios.

Some network caveats may arise when using the current implementation of the real-time data stream, as the receiver needs to have a port open for the connection to be established. This is a problem on *WAN* networks, as the usage of a *network address translation* or *NAT* is very extended, and may lead to not being able to receive external connections. Multiple solutions could be used, but the most reliable is having a server that both end devices can connect to, and forward their messages as needed. Another option is using the MQTT broker as the middle server, but this option should be revised further, as it comes with overhead and the throughput is to be analyzed and measured. For this project's general use-case, all data should always be on the local network, but the possibility of a future request of external access may be impacted due to this problem.

4.5.5.- Firebase alert forwarder improvements

As of the end of this project, the *alert forwarder* developed in Python has all configuration done statically on its source files. Some general cleanup should be done to obtain parameters through the environment or configuration files. A more important isssue is Firebase *tokens*. Those should be able to be updated dynamically, without having to restart the script entirely. To support multiple devices, the program needs to keep up with a mapping of the currently registered ESP32 devices, and the Firebase *tokens* that it should notify on alert. This configuration could be easily done through a specific topic via MQTT.

Security standard practices should be followed and some type of access control to this parameters have to be implemented.

4.5.6.- Access control

As well as the access control for the *alert forwarder*, the ESP32 RPC could also benefit from some type of permissions. This can be easily done via the MQTT broker configuration, as the *mosquitto* server used allows for defining users and passwords to log into the server. This users can have access control rules that define which topics they are allowed to subscribe to, and to which ones they can publish. A special *admin* user could publish to RPC and the *alert forwarder* configuration, but a *standard* user may only see periodic data and alerts of their devices.

4.5.7.- Improved IMU alerts

The proof of concept only allows basic thresholds to be set for the IMU parameters, as well as the calculated magnitude of the acceleration vector. Future developments could include a more flexible alert system for this sensor.

For example, one simple implementation could be analyzing the vibration pattern. This can be seen as a double threshold: one for the ”count this as a vibration”, and another for ”how many vibrations shall occur in a given timeframe to alert”. A patient could have a little ”twitch” or sudden movement, without signifying an emergency scenario. This can happen multiple times over a long period of time, but if it happens too quickly, an alarm could be sent. This can also be extrapolated to a more general use of the system: If you want to monitor patients that are not movement-restrained, any fast spin while sleeping to reposition may trigger the basic thresholds.

Another way of implementing improved alerts is via a machine learning algorithm running on the ESP32. By analyzing historic data, a machine learning algorithm could be developed that, given the real-time acceleration and information from the other sensors, can predict whether the patient is suffering an emergency condition. This is a more advanced way of generating alerts than the threshold, and may not be as flexible, but it can be adjusted and fine-tuned to be more precise for specific cases, which can be critical to the survival of the patient.

4.5.8.- Data visualization

The *telegraf* container is saving all periodic measures into a database. The only way to visualize this data is to query the database directly. A front-end should be developed that allows users to explore and export historic data of specific devices, giving an easier way to professionals to analyze the data.

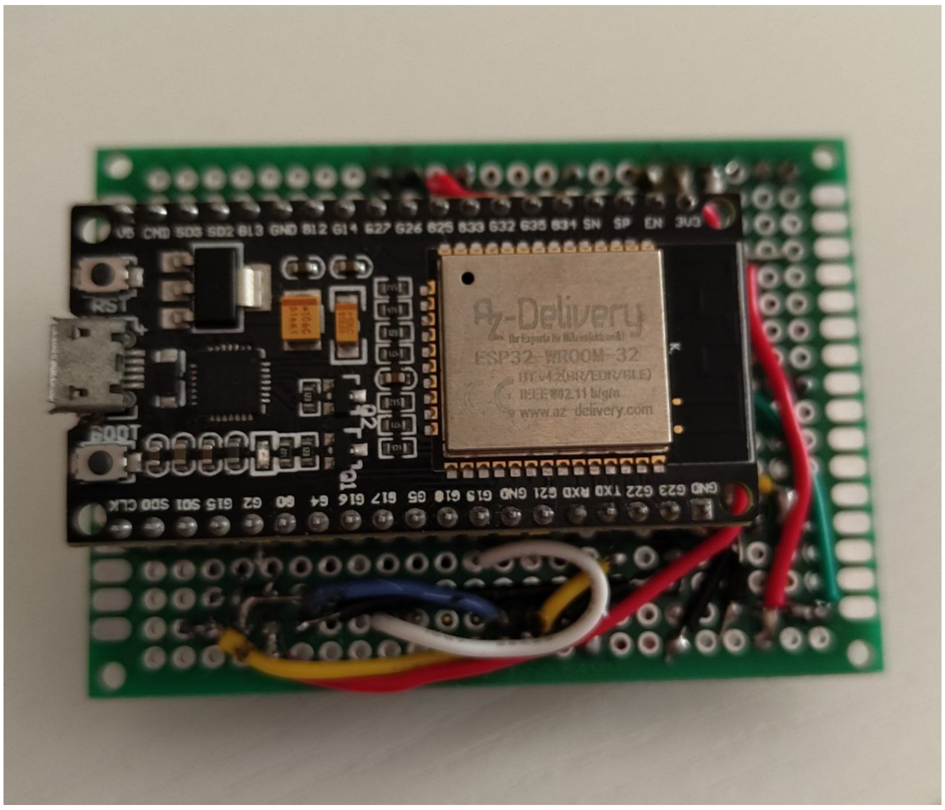

Figure 4.8: The underside of the board, with the ESP32 module.

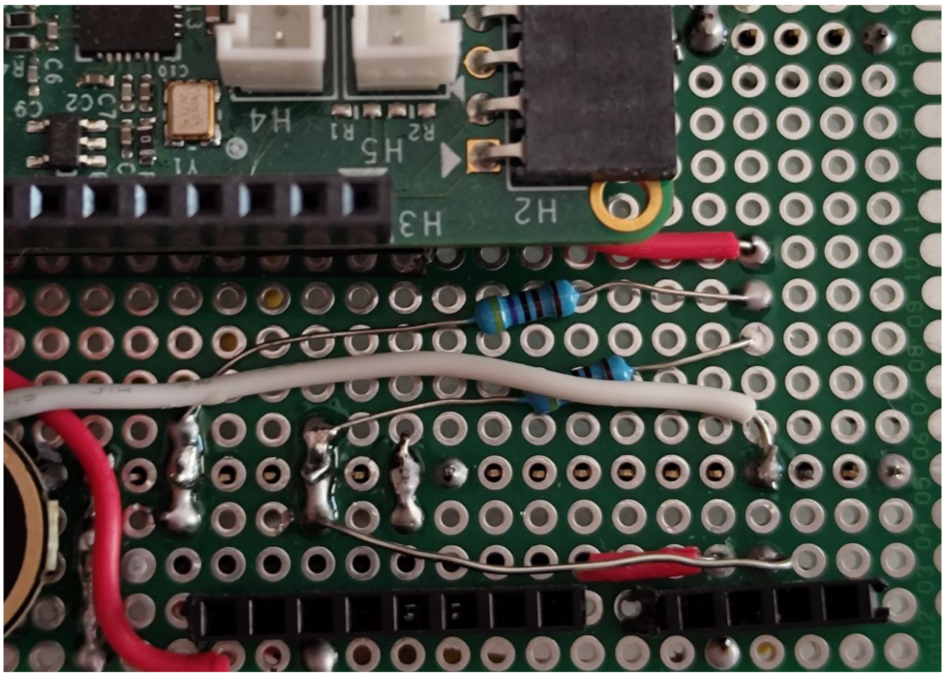

Figure 4.9: Pull-up resistors on the data and clock lines of the *I2C* bus.

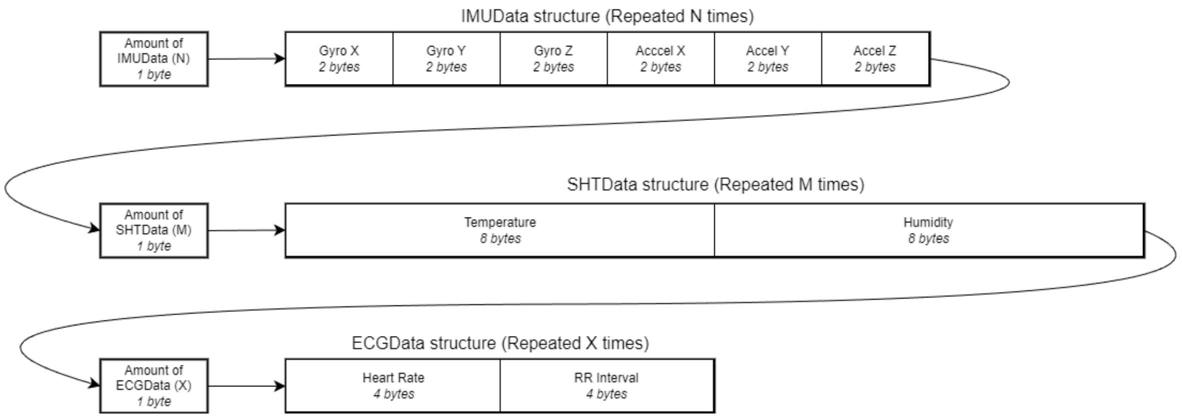
IMUData structure (Repeated N times)
Amount of IMUData (N) 1 byte
Gyro X 2 bytes
Gyro Y 2 bytes
Gyro Z 2 bytes
Accel X 2 bytes
Accel Y 2 bytes
Accel Z 2 bytes
SHTData structure (Repeated M times)
Amount of SHTData (M) 1 byte
Temperature 8 bytes
Humidity 8 bytes
ECGData structure (Repeated X times)
Amount of ECGData (X) 1 byte
Heart Rate 4 bytes
RR Interval 4 bytes

Figure 4.10: Simplified diagram with a sample packet including 1 of each data structure.

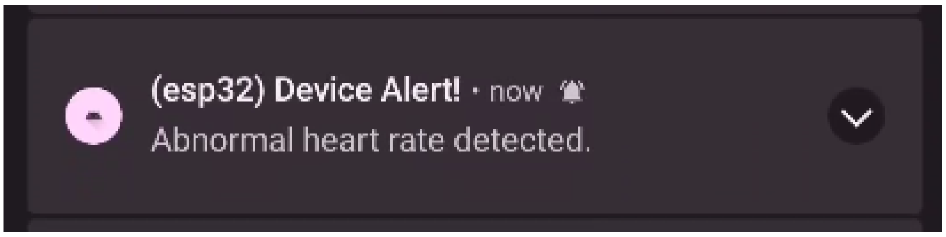
(esp32) Device Alert! · now
Abnormal heart rate detected.

Figure 4.11: Sample notification from a heart rate alert.

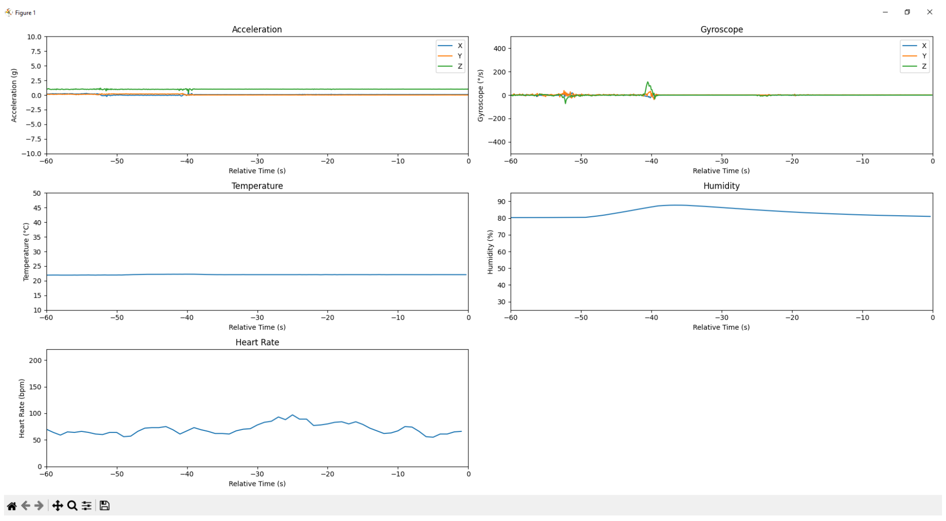

Figure 4.12: Sample GUI image plotting data of an idle subject.

5. Technical documentation

5.1.- User Requirements

ID	Name	Description
UR1	**Body parameter monitoring**	
UR1.1	Parameters to measure	The user needs to monitor the following parameters from the patient over a defined period: Heart Rate, Body Humidity, Body Temperature, Movement, Noise.
UR1.2	Modifiable report period	The user will be able to modify the period in which the measures are sent.
UR1.3	On-Demand real-time view of parameters	The user will be able to view the measured parameters in real-time, if needed.
UR2	**Alerts and notifications**	
UR2.1	Patient health alert	The user will receive a phone notification when the body parameters of the patient are within a defined threshold.
UR2.2	Modifiable thresholds	The user will be able to modify the thresholds taken into account for notifications.
UR3	**Data storage and access**	
UR3.1	Storage of periodic measures	The periodic measures will be stored safely and the user shall be in full possession of them.
UR3.2	Storage of real-time measures	The real-time measures will be stored safely and the user shall be in full possession of them.
UR3.3	Visualization of stored measures	The user will be able to observe the measures.
UR3.4	Deletion of stored measures	The user will be able to safely delete the measures.
UR4	**Remote management and configuration**	
UR4.1	Remote configuration	The user will be able to modify the configuration remotely.

ID	Name	Description
UR4.2	Remote operation	The user will be able to control the devices remotely. This includes restarting the device, and enabling or disabling real-time measure report features.
UR4.3	Remote updates	The user will be able to update the devices firmware remotely.
UR5	**Compliance and standards**	
UR5.1	Open source dependencies	The system will use open source dependencies whenever possible.
UR5.2	Use of standards	The system will adhere to standards in every module, to ensure maintainability and compatibility.
UR5.3	Safe for medical use	The device will follow medical safety precautions, always ensuring patient well-being, as well as their surroundings.

Table 5.1: User Requirements

5.2.- System Requirements

ID	Name	Description	User Requirement
—	**Functional Requirements**		
FR1	Parameters to measure	The device will monitor and periodically report the following parameters from the patient: Heart Rate, Body Humidity, Body Temperature, Movement, Noise. The data will be sent out of the device via *WiFi*, over *MQTT*, on a defined topic.	**UR1.1**
FR2	Modifiable report period	The device will accept new configuration on its *RPC* topic over *MQTT*. This configuration has a field that changes the interval in which data is periodically reported.	**UR1.2, UR4.1**

ID	Name	Description	User Requirement
FR3	On-Demand real-time view of parameters	The device will start a real-time transmission of sensor data or audio data when a specific *RPC* call is received over *MQTT*. This data shall not be sent over *MQTT*, as to increase the data throughput and reduce latency.	**UR1.2, UR4.2**
FR4	Patient health alert	The device will send an alert on a specific *MQTT* topic when a defined threshold is reached for any sensor. The *alert handler* will check the phone *tokens* assigned to that device, and forward this alert to *FCM*, in order to show a notification on said phones.	**UR2.1**
FR5	Modifiable thresholds	The device will accept new configuration on its *RPC* topic over *MQTT*. This configuration has four fields *(min, low, high, max)* for each sensor, so multiple cases can be covered with a single configuration.	**UR2.2, UR4.1**
FR6	Storage of periodic measures	The system will store the periodically reported measures from devices into a local database.	**UR3.1**
FR7	Storage of real-time measures	The system will store the real-time measures from devices into a local database.	**UR3.2**
FR8	Visualization of stored measures	The system will allow the user to see, filter and extract data from the local database. Periodic measures will be separated from real-time data.	**UR3.3**
FR9	Deletion of stored measures	The system will allow the user to delete data from the local database.	**UR3.4**
FR10	Remote configuration	The device will accept new configuration on its *RPC* topic over *MQTT*. This configuration must be saved on the device file system, for persistence.	**UR4.1**
FR11	Remote operation	The device will accept *RPC* calls on its specific topic over *MQTT*. This includes. but is not limited to: restarting the device, starting or stopping real-time data reporting.	**UR4.2**

ID	Name	Description	User Requirement
FR12	Remote updates	The device will accept an *RPC* call with a URL in which the *OTA* update binary is hosted. Then, it must download the file and perform the update, rolling back the version if an error is found.	**UR4.3**
FR13	Open source dependencies	The device and supporting components will use open source libraries, frameworks and tool-chains on its code. Some closed source code is executed due to the necessity of *FCM* and Android notifications.	**UR5.1**
FR14	Use of standards	The device and supporting components will adhere to standards on their specific domain.	**UR5.2**
FR15	Safe for medical use	The device will follow medical safety precautions, and guidance will be given for proper usage.	**UR5.3**
—	**Non Functional Requirements**		
NFR1	Usability	The device should be minimally invasive, and comfortable for long-term wear. The overall system should be easy to use, given a basic guidance.	
NFR2	Maintainability	The system shall adhere to open standards and frameworks as much as possible. The devices should be easy to maintain and replace. Firmware updates and recalibration must be possible remotely.	
NFR3	Scalability	The system must accept multiple devices and phones simultaneously. The devices must be suited for future functionality updates without replacement.	
NFR4	Performance	The devices should be able to operate in real-time. The alerts must arrive quickly to phones without any delays.	

ID	Name	Description	User Requirement
NFR5	Reliability and Availability	The overall system must be reliable and have great availability, to ensure no alerts are missed. The cost of parts and modules used on devices should be minimized, but they must be tested and generally available for purchase.	
NFR6	Security and Safety	Given the nature of the data obtained, the system must adhere to strict security norms to ensure patient data protection. The device must comply with the needed safety precautions.	

Table 5.2: System Requirements

5.3.- Technical Manual

The development environment is comprised of a Windows machine for general programming, a phone running Android, and a Raspberry Pi 4 running Linux for the needed services. Any similar configurations (such as using an Android emulator) may follow the same steps as this manual. This manual assumes a standard installation of Windows and Linux are done on their respective systems, and assumes the user can control both with no issues.

The first step is to get the IDE ready. Visual Studio Code can be downloaded and installed at `https://code.visualstudio.com/Download`. Upon installation, the PlatformIO IDE extension can be installed from inside Visual Studio Code, by searching from it in the extension marketplace from the extensions menu *(Ctrl+Shift+X)*. The installation may take a while, as multiple build dependencies are installed.

The project's source (*source.zip*) code shall be decompressed into a folder, which will be considered the root folder. This folder can be opened in Visual Studio Code for PlatformIO to recognize the project. After that, the program can easily be built and uploaded to a connected ESP32 from the PlatformIO menu on the left (the alien-looking icon). The connected ESP32 has to be in the bootloader; to access it, connect the ESP32 development board while holding the *boot* button, or if it is already connected, hold the *boot* button and press the *reset/RST* button. Any changes made to the project have to be uploaded into the ESP32 this way.

A sample of the entire ESP32 SPIFFS filesystem can be found under the `data/` directory of the project. To upload this into the SPIFFS partition of the ESP32, connect/put it in bootloader mode, and select ”Upload Filesystem Image” on the PlatformIO menu.

In order to set up the containers running on the Raspberry Pi, *docker* will need to be installed. First, the *apt* sources shall be updated, and any packages too. Then, docker can be installed using the convenience script. Adding the `pi` user to the `docker` group is not mandatory, but very useful for later commands, to remove the need of using `sudo` every time. The following snippet (18) shows how to do this steps:

```
sudo apt update
sudo apt upgrade

sudo curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

sudo usermod -aG docker pi
```

Listing 18: Commands used to install docker onto a fresh Raspberry Pi.

A reboot should be done before continuing, to ensure everything is installed correctly, and the group change for the `pi` user is effective. One of the many commands to reboot is `sudo reboot now`. After that, the project’s source zip file can be extracted onto a folder. The `Docker/` folder contains the *docker compose* files to start up the *mosquitto* MQTT broker and the *telegraf* metrics collector.

First, create the required network by running `docker network create tfg`. This should only be done once. After that, navigate to their respective folders, and execute `docker compose up -d`. It may take a while, as it downloads the images needed for the containers. At the end, both should be up and running, which can be verified by running `docker ps`. The following snippet of code (Listing 19) shows an example, given the source zip file *source.zip*.

```
unzip source.zip -d tfgproject

docker network create tfg
cd tfgproject/Docker/mosquitto
docker compose up -d
cd ../telegraf
docker compose up -d

docker ps
```

Listing 19: Commands used to setup the containers for the first time.

After this, the services should start up whenever the Pi boots up. Only the Firebase *alert forwarder* is left as the only critical component. Ensure the Python interpreter, as well as the package manager *pip* and the virtual environment utility *venv* are installed with this command: `sudo apt install python3 python3-pip python3-venv`. Now, in order to create the needed virtual environments, navigate to the project's root directory, and enter the `supporting_python/firebase/` folder. On there, the environment should be created under the `.venv` directory name, and the library requirements from `requirements.txt` shall be installed. The following commands (Listing 20) show how it should be done:

```
python3 -m venv .venv
.venv/bin/python -m pip install -r requirements.txt
```

Listing 20: Commands used to setup a Python virtual environment.

Finally, lets configure *cron* so that the Firebase starter script is called at boot. Ensure the file is executable by running `chmod +x setup.sh`, and then open the *crontab* with `crontab -e`. If it asks for an editor, just press enter to use the default *nano* editor. The crontab will open in the editor. Navigate to the end of the file and add a line like this:

```
@reboot /home/pi/tfgproject/supporting_python/firebase/start.sh
```

Now, upon reboot, all three components will be running.

Finally, for the notifications to work, the *NotificationsTFG* app needs to be installed on a device and the Firebase *token* should be saved.

Install Android Studio on the Windows development machine following the instructions at `https://developer.android.com/studio/install`.

Open the `android_source.zip` in Android Studio and install the app on the phone or emulator (or install the *apk* file). After accepting the notification permissions, the screen and the *logcat* logs will show the firebase *token*. Add it to the list of tokens of the firebase *alert forwarder* by modifying the `FIREBASE_TOKENS` list of `mqtt.py`, inside the Raspberry Pi project folder. Restart the Raspberry Pi. You may now close the app. Now, any new alerts will also be sent to that phone.

6. Conclusions

The end result is a working proof of concept that sends alerts as phone notifications every time the configured threshold is reached for the sensor parameters. It has quality of life features to configure and reprogram the microcontroller remotely. In order to use it as a final product, some extra development is still needed, such as hardware design, but the baseline is set and proven to work correctly. Data can be analyzed and recorded for later improvements, which can be implemented without physical intervention thanks to the OTA firmware update functionality. Overall, the project is in a modular and stable state, ready to be improved and expanded easily.

Some of the developer experience could be simplified by having more automated tools, but given the amount of modularity of the system, and the minimum of three devices, it is still cumbersome to work on every side of the project at once. Multiple trade-offs had to be done. Nevertheless, the experience and acquired knowledge about all the different topics was very broad and entertaining, and the end result is a very big step towards a useful tool.

Currently, it can be used as a way to record some extended test data, but having the board close to the patient skin is not safe nor comfortable. As such, the more meaningful data that could be acquired safely are the measures reported by the IMU, as well as the audio. The heart rate can also be acquired safely through the modified band connector, but some value errors are to be expected, as they have appeared in current tests.

This project can be expanded into a functional medical product by the addition of the future work outlined in Section 4.5, which could help future patients to have a better experience and care during their therapy.

6.1.- Sample graphs and results

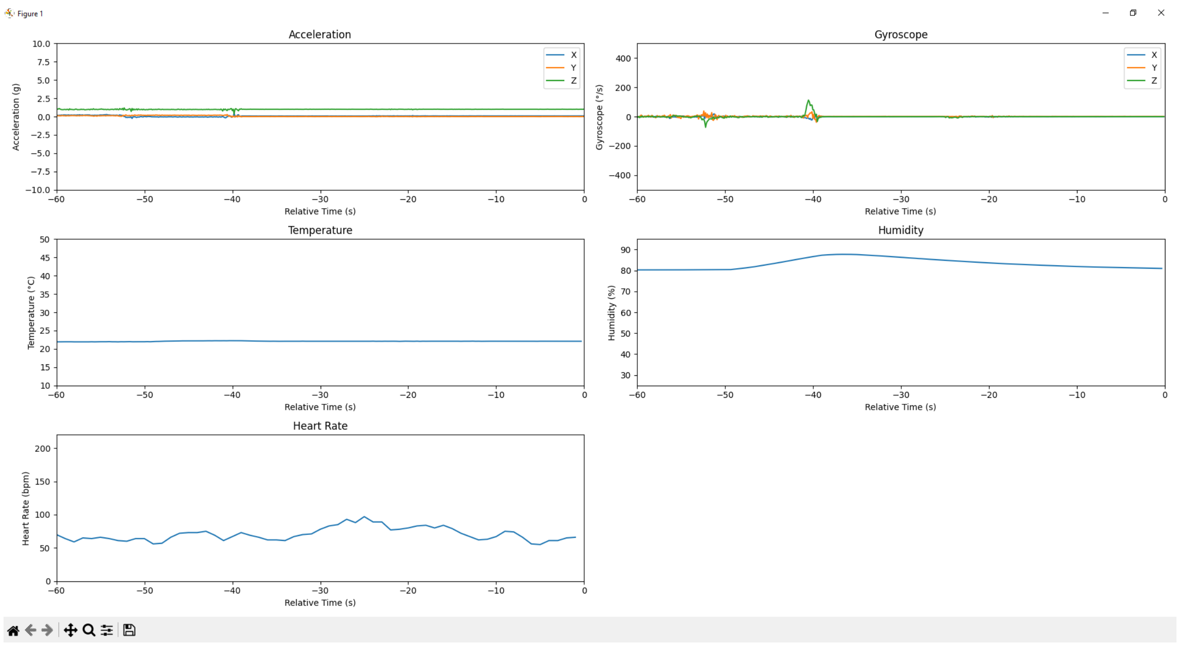

Figure 6.1: Parameters of an idle/resting test subject.

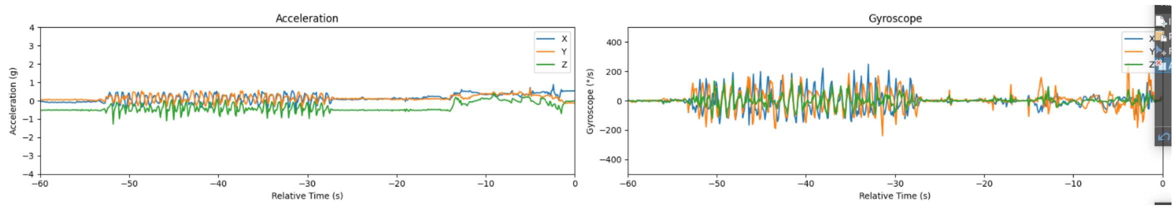

Figure 6.2: Acceleration and gyroscope values of a test subject simulating seizure shaking.

7. Annex

7.1.- ESP32 initialization sequence diagram

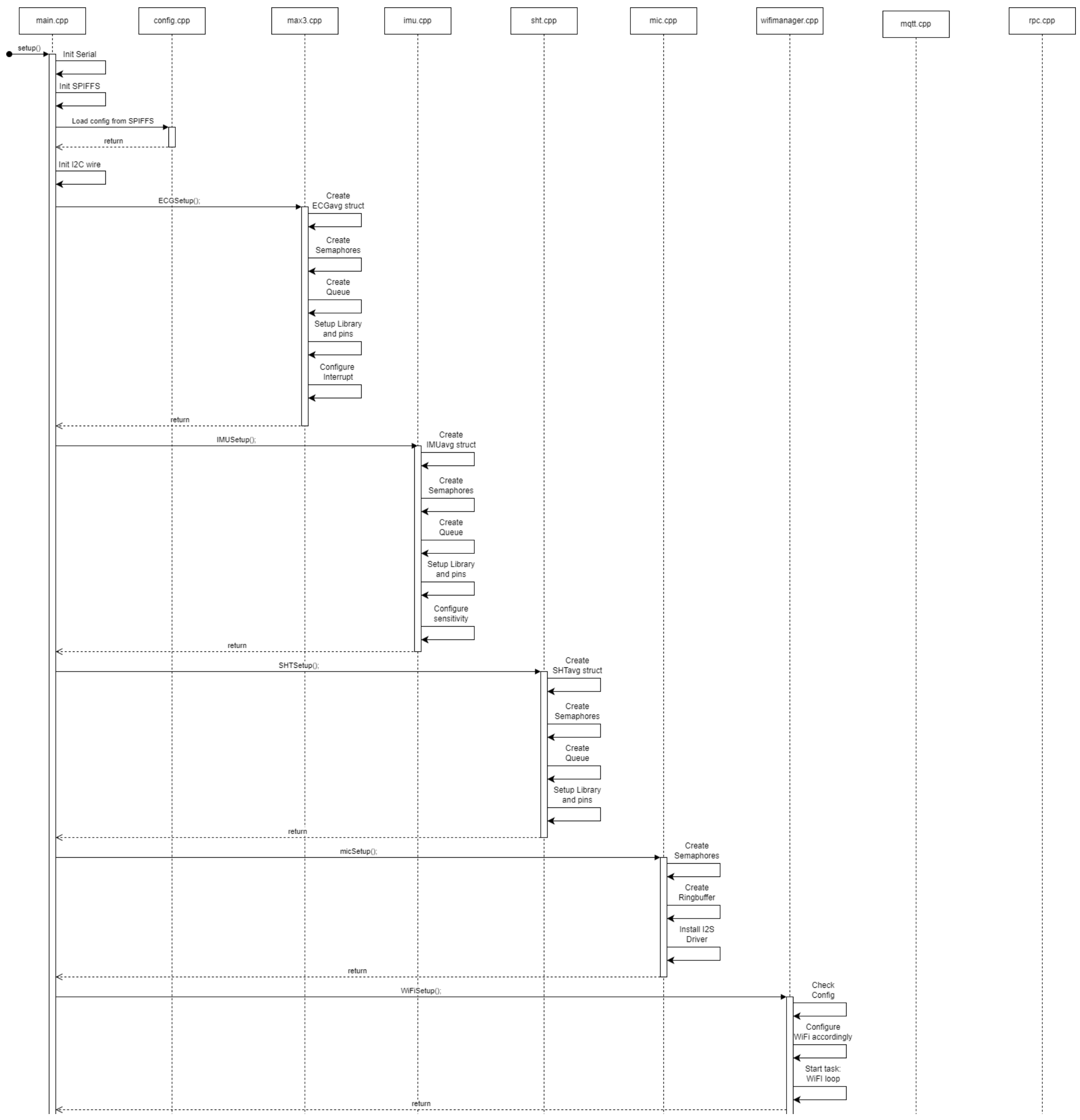

Figure 7.1: ESP32 initialization sequence diagram (part 1/2).

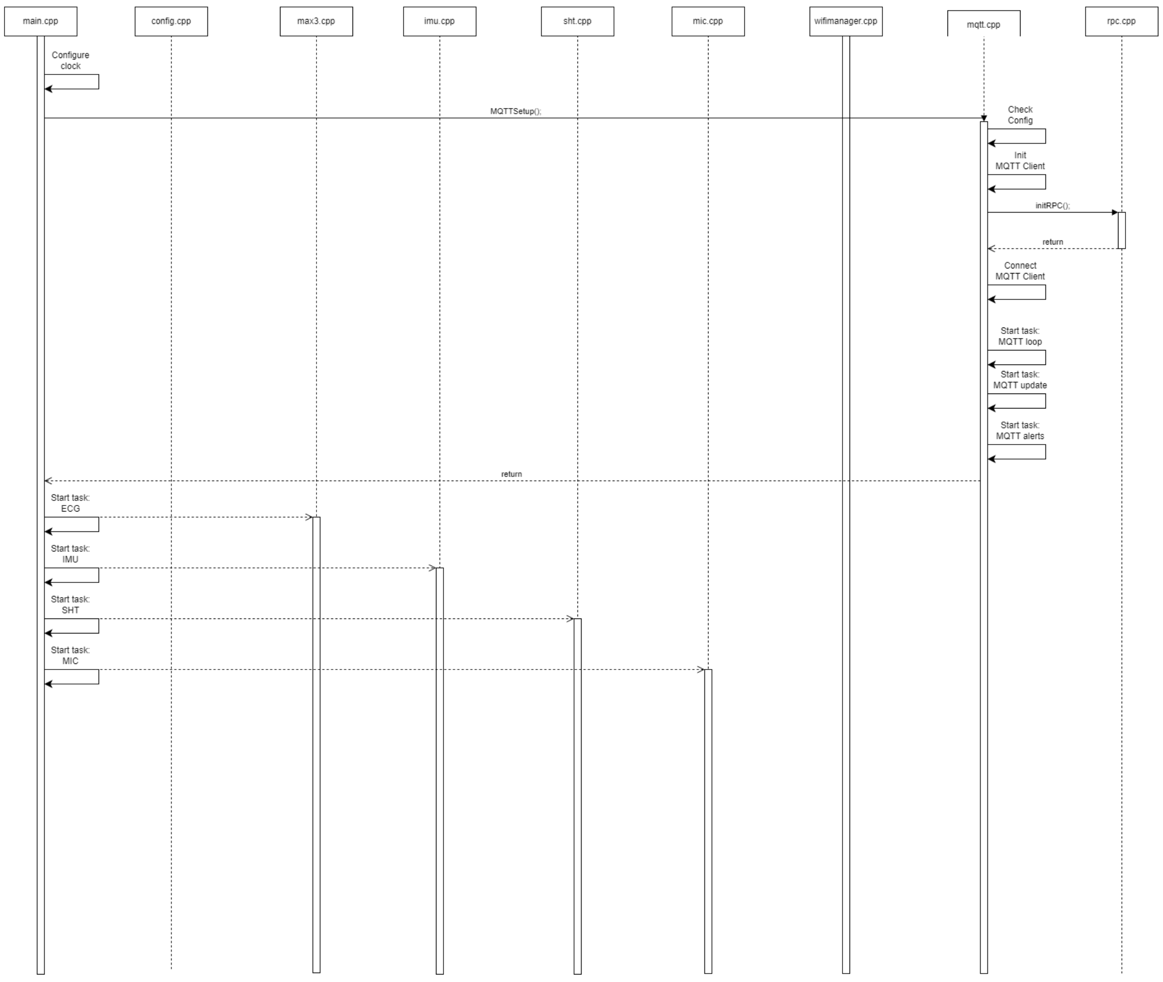

Figure 7.2: ESP32 initialization sequence diagram (part 2/2).

Bibliography

[1] Arduino MKR WiFi 1010, . URL https://store.arduino.cc/products/arduino-mkr-wifi-1010. Accessed on July 2023.

[2] Arduino Uno Rev.3, . URL https://store.arduino.cc/collections/boards/products/arduino-uno-rev3. Accessed on July 2023.

[3] Mongoose OS - an IoT integrated development framework. URL https://mongoose-os.com/mos.html. Accessed on July 2023.

[4] Amazon Web Services. AWS IoT Core pricing. URL https://aws.amazon.com/iot-core/pricing/. Accessed on July 2024.

[5] Analog Devices Inc. Max30003 ultra-low power, single-channel integrated biopotential (ECG, R-to-R detection) AFE datasheet. URL https://www.analog.com/media/en/technical-documentation/data-sheets/max30003.pdf. Accessed on January 2023.

[6] Anthony Atkielski. ECG of a heart in normal sinus rhythm. URL https://commons.wikimedia.org/wiki/File:SinusRhythmLabels.svg. Accessed on July 2023.

[7] Benoît Blanchon. Why must I create a separate config object? URL https://arduinojson.org/v6/faq/why-must-i-create-a-separate-config-object/. Accessed on July 2024.

[8] Piyu Dhaker. Introduction to SPI interface, 2018. URL https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html#author. Accessed on July 2023.

[9] Docker Inc. and collaborators. Docker compose overview, 2024. URL https://docs.docker.com/compose/. Accessed on July 2024.

[10] Earle F. Philhower. Arduino-Pico. URL https://github.com/earlephilhower/arduino-pico. Accessed on July 2023.

[11] Espressif Systems. ESP32 documentation: Inter-IC Sound (I2S), . URL https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/i2s.html. Accessed on July 2024.

[12] Espressif Systems. *ESP32-WROOM-32 Datasheet*, . URL https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf. Accessed on July 2023.

[13] Espressif Systems. *ESP8266EX Datasheet*. . URL https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf. Accessed on July 2023.

[14] Espressif Systems. GPIO matrix and pin mux, 2022. URL https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/tutorials/io_mux.html. Accessed on July 2023.

[15] Espressif Systems. SMP on an ESP target, 2023. URL https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos_idf.html. Accessed on July 2024.

[16] Espressif Systems. FreeRTOS (supplemental features), 2023. URL https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos_additions.html. Accessed on July 2024.

[17] Firebase. FCM architectural overview., . URL https://firebase.google.com/docs/cloud-messaging/fcm-architecture. Accessed on June 2024.

[18] Firebase. Firebase Documentation: Send messages to device groups on Android, . URL https://firebase.google.com/docs/cloud-messaging/android/device-group. Accessed on July 2024.

[19] Firebase. Firebase Documentation: Recieve messages on an Android app. handling messages, . URL https://firebase.google.com/docs/cloud-messaging/android/receive#handling_messages. Accessed on June 2024.

[20] Firebase. Server side FCM sample, . URL https://github.com/firebase/quickstart-python/blob/master/messaging/messaging.py. Accessed on June 2024.

[21] Sam Gross. Making the Global Interpreter Lock Optional in CPython, 2023. URL https://peps.python.org/pep-0703. Accessed on July 2024.

[22] Gareth Halfacree and Ben Everard. *Get Started with MicroPython on Raspberry Pi Pico*. Raspberri Pi Press, 2021. URL https://store.rpipress.cc/products/get-started-with-micropython-on-raspberry-pi-pico. Accessed on July 2023.

[23] Muhammad Husaini, Latifah Munirah Kamarudin, Ammar Zakaria, Intan Kartika Kamarudin, Muhammad Amin Ibrahim, Hiromitsu Nishizaki, Masahiro Toyoura, and Xiaoyang Mao. Non-contact breathing monitoring using sleep breathing detection algorithm (sbda) based on uwb radar sensors. *Sensors*, 22(14), 2022. ISSN 1424-8220. doi: 10.3390/s22145249. URL https://www.mdpi.com/1424-8220/22/14/5249.

[24] Invensense Inc. *MPU6050 Datasheet*. 2013. URL https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. Accessed on June 2024.

[25] InvenSense Inc. *INMP441 Datasheet*. 2014. URL https://invensense.tdk.com/wp-content/uploads/2015/02/INMP441.pdf. Accessed on June 2024.

[26] David Johnson-Davies. μlisp - Lisp for microcontrollers. URL http://www.ulisp.com/show?3M. Accessed on July 2023.

[27] JSON-RPC Working Group. JSON-RPC 2.0 Specification, 2013. URL https://www.jsonrpc.org/specification. Accessed on July 2024.

[28] Kubii. Buy Raspberry Pi Pico W from Kubii, official retailer. URL https://www.kubii.com/es/las-tarjetas-raspberry-pi/3205-1641-raspberry-pi-pico-w-h-wh-3272496311589.html#/version_pico-pico_w. Accessed on July 2023.

[29] David Kushner. The Making of Arduino. 2011. URL https://spectrum.ieee.org/the-making-of-arduino. Accessed on July 2023.

[30] Maxim Integrated. MAX30003WING expansion board. URL https://www.analog.com/media/en/technical-documentation/data-sheets/MAX30003WING.pdf. Accessed on January 2023.

[31] Renzo Mischianti. ESP32-WROOM-32 Module Pinout. URL https://mischianti.org/2021/05/26/esp32-wroom-32-high-resolution-pinout-and-specs/. Accessed on July 2023.

[32] Mouser. Buy ESP32-WROOM-32E from Mouser. URL https://mou.sr/46QwNy4. Accessed on July 2023.

[33] Rubén Muñiz, Juan Díaz, Juan A. Martínez, Fernando Nuño, Julio Bobes, Mª Paz García-Portilla, and Pilar A. Sáiz. A smart band for automatic supervision of restrained patients in a hospital environment. *Sensors*, 20(18), 2020. ISSN 1424-8220. doi: 10.3390/s20185211. URL https://www.mdpi.com/1424-8220/20/18/5211.

[34] OASIS MQTT Technical Committee. MQTT version 3.1.1, 2014. URL https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Accessed on July 2024.

[35] OSHWA (Open Source HardWare Association). A resolution to redefine SPI signal names, 2022. URL https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/. Accessed on July 2023.

[36] Oxford Medical Education. ECG interpretation. URL https://oxfordmedicaleducation.com/ecgs/ecg-interpretation/. Accessed on July 2023.

[37] Jiapu Pan and Willis J. Tompkins. A real-time qrs detection algorithm. *IEEE Transactions on Biomedical Engineering*, BME-32(3):230–236, 1985. doi: 10.1109/TBME.1985.325532.

[38] PlatformIO. PlatformIO registry. URL https://registry.platformio.org/. Accessed on July 2024.

[39] Raspberry Pi Foundation. *Raspberry Pi Pico Pinout*. URL https://www.raspberrypi.com/documentation/microcontrollers/images/pico-pinout.svg. Accessed on July 2023.

[40] Raspberry Pi Foundation. Raspberry pi pico datasheet, 2023. URL https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf. Accessed on July 2023.

[41] Raspberry Pi Foundation. *RP2040 Datasheet*, 2023. URL https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf. Accessed on July 2023.

[42] Marco Polo Sauza Aguirre. Breathing sensor 1 respiration sensor. URL https://www.hackster.io/marco-polo-sauza-aguirre/breathing-sensor-1-respiration-sensor-1b18de. Accessed on March 2023.

[43] Sensirion. *SHT21 Datasheet*. 2022. URL https://sensirion.com/media/documents/120BBE4C/63500094/Sensirion_Datasheet_Humidity_Sensor_SHT21.pdf. Accessed on June 2024.

[44] STMicroelectronics. STM32 32-bit Microprocessors, . URL https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html. Accessed on July 2023.

[45] STMicroelectronics. Buy STM32 wireless MCUs, . URL https://estore.st.com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-wireless-mcus.html. Accessed on July 2023.

[46] STMicroelectronics. Buy STM32WBA52CEU6, . URL https://estore.st.com/en/stm32wba52ceu6-cpn.html. Accessed on July 2023.

[47] Stephanie Susnjara and Ian Smalley. What is Docker?, 2024. URL https://www.ibm.com/topics/docker. Accessed on July 2024.

[48] Espressif Systems. *ESP32 Datasheet*. URL `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`. Accessed on July 2023.

[49] Texas Instruments Inc. *A basic guide to I2C*. 2022. URL `https://www.ti.com/lit/an/sbaa565/sbaa565.pdf`. Accessed on June 2024.

[50] uPesy. How to use the gpio pins of the ESP32, 2022. URL `https://www.upesy.com/blogs/tutorials/esp32-pinout-reference-gpio-pins-ultimate-guide`. Accessed on July 2023.