POLITECNICO DI TORINO

Master's degree course
in Electronics for Industrial Applications

Master Thesis

Development of a wearable device for the acquisition of a six-derivations ECG for self-monitoring of cardiovascular diseases.

Supervisors
prof. Eros Gian Alessandro Pasero
Co-supervisors
prof. Vincenzo Randazzo

Candidates
Marco Sento
Gianmarco Dogliani

Academic year 2022-2023

"The beauty and the scent of roses can be used as a medicine and the sun rays as a food."
— Nikola Tesla

Contents

Chapter 1

Introduction

Developed over a century ago, the electrocardiogram provides invaluable information about the state of health of the heart, aiding in the diagnosis and management of various cardiovascular conditions.
Cardiovascular diseases (CVDs) have significant impacts on health, quality of life and economics, being the major cause of death and disability worldwide.
Also in Europe, CVDs are the leading cause of death, accounting for 32% of all deceases with an estimated cost of €210 billion annually.
Addressing CVDs through prevention and monitoring is crucial to improve health expectancy and reducing the economic burden of these diseases.

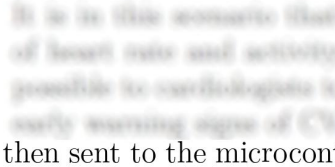
It is in this scenario that wearable smart devices fit: they allow self-monitoring of heart rate and activity, and other vital signs, providing feedback and making possible to cardiologists to remotely monitor and cure patients, allowing to detect early warning signs of CVDs and to intervene before complications occur.
There are several devices that allow self monitoring of heart activity but they are not designed to provide a comprehensive diagnosis of cardiac conditions. They commonly provide just numerical information about heart activity and, less frequently, reports a single-lead ECG.

This thesis has the aims to study and to realise a novel wearable device capable to overcome these limitations, delivering a six derivations medical ECG in a non-invasive way.

While a single-lead ECG offers a reduced diagnostic utility, showing the electrical activity of the heart from one direction only, a six-lead ECG is more useful for diagnosing cardiac conditions such as arrhythmias, ischaemias, and heart blocks, by recording information from six different electrodes placed on specific locations on chest and limbs.

The device developed achieves these results using an ultra-compact design and employing only three electrodes by leveraging on the mathematical relationship which occurs between the six derivations.

Going more into details of the signal elaboration system, the device acquires two



then sent to the microcontroller and finally, they are transmitted to a smartphone application via Bluetooth interface.
The work carried out also includes a further processing of the signals through digital filtering such as Finite Impulse Response (FIR) filters.

The whole processing gives the possibility to appreciate six different ECG waveforms in real time on the smartphone display, just by placing the fingers on the device.

In general, the development of this appliance has gone through the following steps:

- Design of the different functional blocks of the circuit.
- Board design using OrCad software through schematic diagram and PCB design.
- Development of firmware.
- Implementation of digital signal processing techniques.

To conclude, the combined use of digital and analog filtering of the signals acquired via three electrodes, made it possible to create a device capable of acquiring a high-quality six-derivations electrocardiogram while maintaining an extremely compact design.
By offering an accessible and user-friendly solution, this device has the potential to simplify and change the ECG monitoring as known until now, improving diagnosis, treatment, and prevention strategies for cardiovascular diseases.

1.1 Cardiovascular diseases nowadays

Cardiovascular diseases (CVDs) remain a significant global health concern, causing a substantial burden on individuals, communities, and healthcare systems despite

the advancements in medical research, technology, and interventions.
In recent decades, the prevalence of cardiovascular diseases has steadily increased, becoming the leading cause of mortality worldwide, primarily due to changes in lifestyle and an aging population.
According to the World Health Organization (WHO), CVDs are responsible for approximately 17.9 million deaths each year, accounting for 37% of all the premature deaths: ischemic heart disease, stroke, and heart failure are the most common cardiovascular conditions.
Several modifiable and non-modifiable risk factors contribute to the development of cardiovascular diseases: non-modifiable risk factors include age, gender, and family history, while modifiable risk factors encompass unhealthy diet, physical inactivity, tobacco use, obesity, hypertension, diabetes, and high cholesterol levels.

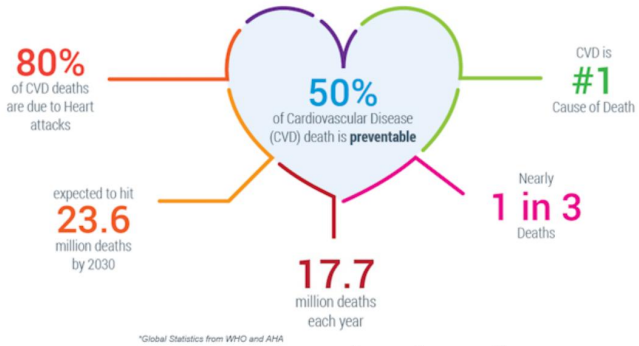


Figure 1.1: Infographic about CVDs.

Early and accurate diagnosis of cardiovascular diseases is essential for effective management and prevention of complications. Traditional diagnostic methods, such as electrocardiography (ECG), echocardiography, and stress tests, continue to play a vital role, however, advancements in technology have introduced novel diagnostic tools, including cardiac imaging techniques like computed tomography and magnetic resonance.
In addition, wearable devices and smartphone apps have gained popularity in heart health monitoring nowadays.
As can be seen, the management of cardiovascular diseases encompasses a multifaceted approach, involving lifestyle modifications, pharmacological interventions, and invasive procedures.
In any case, lifestyle behaviours including a heart-healthy diet, regular exercise, smoking cessation and stress reduction are essential to prevent and manage CVDs.

1.2 Prevention and technology

Advancements in technology continue to revolutionize the field of cardiovascular medicine: wearable devices, such as smartwatches and fitness trackers, provide real-time monitoring of heart rate, activity levels, and ECG data, enabling early detection of abnormalities.
Telemedicine and remote patient monitoring have also gained attention, facilitating access to specialized care and improving patient outcomes, particularly in rural or underserved areas.
Despite significant progress in the prevention, diagnosis, and management of cardiovascular diseases, several challenges persist.
Access to quality healthcare, particularly in low-resource settings, remains a concern, additionally, the rising prevalence of risk factors, such as obesity and aging, poses ongoing challenges for disease prevention.
By identifying and addressing risk factors early on, the incidence and severity of CVDs can be significantly reduced.
Prevention encompasses both population-wide strategies and individual-level interventions, in this context public health campaigns and education aim to promote healthier lifestyles, while personalized risk assessments and targeted interventions provide tailored approaches to high-risk individuals.

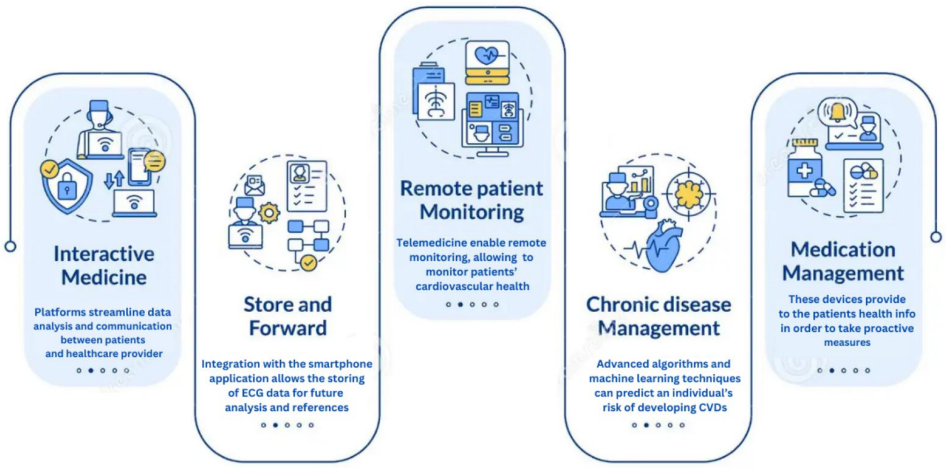


Figure 1.2: Infographic about telemedicine.

Technology has revolutionized the field of cardiovascular prevention, providing new tools and approaches for risk assessment, monitoring, and interventions.
There are many fields in which technology is having an enormous impact, profoundly affecting the treatment of cardiovascular diseases:

a. **Risk Assessment**: Advanced algorithms and machine learning techniques can analyze vast amounts of data, including genetic information, medical history, and lifestyle factors, to predict an individual's risk of developing CVDs. This personalized risk assessment enables early interventions and targeted prevention strategies.

b. **Wearable Devices**: Smartwatches, fitness trackers, and other wearable devices allow continuous monitoring of heart rate, activity levels, sleep patterns, and ECG data. These devices provide valuable insights into an individual's cardiovascular health, empowering them to make informed decisions and take proactive measures.

c. **Telemedicine and Remote Monitoring**: Telemedicine platforms enable remote consultations, allowing healthcare providers to monitor patients' cardiovascular health, provide guidance, and adjust treatment plans without the need for in-person visits. Remote patient monitoring further enhances this approach by continuously collecting and transmitting vital signs and other relevant data, ensuring timely interventions and reducing the risk of complications.

d. **Mobile Applications**: Smartphone applications offer various functionalities, such as diet and exercise tracking, medication reminders, and stress management tools. These apps promote lifestyle modifications, facilitate self-monitoring, and provide educational resources to promote cardiovascular health.

e. **Digital Health Platforms**: Integrated digital health platforms bring together various technologies, including wearable devices, mobile apps, and electronic health records. These platforms streamline data collection, analysis, and communication between patients and healthcare providers, facilitating personalized prevention plans and enhancing care coordination.

It is in this scenario that the device realised in this thesis project fits.
It allows instantaneous monitoring of the health status of the patient's heart through the real-time acquisition of six cardiac signals.
The data thus acquired are managed by a smartphone application so that the patient's state of health can be immediately visualised and shared with his or her doctor, paving the way for a new method of telemedicine.

During the development of the product, the manufacturing process was divided into several phases, each of which involved a specific set of operations and choices. To conclude this introduction, a detailed list of the steps taken during the development of our product is reported:

1. Definition of project name and scope;
2. Definition of project specifications;
3. Market analysis of potential competitors;
4. Signal conditioning design;
5. Digital section design;
6. Selection of components;
7. Schematic diagram realized with OrCad - Capture CIS;
8. PCB design using OrCad - Allegro PCB Editor;
9. BOM-Based evaluation of costs;
10. Development of firmware for the STM32WB55RG evaluation board;
11. Processing of the signals acquired with the evaluation board;
12. Prototype development;
13. Electrical check of connections;
14. Mock-up development;
15. Communication with the Android App.

Chapter 2

Electrocardiography: an overview

The electrocardiography has a long history that begins two centuries ago.
It was the late 19^{th} century when the first studies concerning the electrical activity of the heart began. In 1887 the British physiologist Augustus Waller recorded the first human electrocardiogram, using a capillary electrometer[1], having observed that the heartbeat produces an electrical signal measurable as a potential difference on the skin.
The results obtained were approximate but soon the development of new techniques increased considerably and only fourteen years later, in 1901, the Dutch physiologist Willem Einthoven developed an ECG machine with extremely accurate results.
The machine was based on a galvanometer capable to amplify the electrical signal and then to return it on paper.
This new technology allowed the clinical study of the electrical activity of the heart and developed a new method to classify heart diseases like arrhythmias and heart blocks.
He also worked on a solution to standardize the electrocadiogram and realized what became the universally adopted ECG representation.
Throughout the following decades the electrocadiogram was further developed and improved, and still nowadays it is an essential tool for identifying and studying heart diseases.

[1] A capillary electrometer uses changes in the surface tension of a electrolyte inside the capillary to detect small variations in electric potential.

2.1 The ECG signal

The electrocardiogram (ECG) is a visual representation of the electrical activity of the heart over time, it represents the variations of electrical potential difference between two points of the human body.

The ECG waveform is composed of several distinct parts, each one represents a different aspect of the heart's electrical activity.
The main segments of an ECG are:

- The P wave, which represents depolarization of the atria; typical duration <80 ms.
- The QRS complex, which represents depolarization of the ventricles; typical duration 80-100 ms.
- The T wave, which represents repolarization of the ventricles; typical duration 160 ms.
- The U wave, which represents papillary muscle repolarization; typical duration <60 ms.

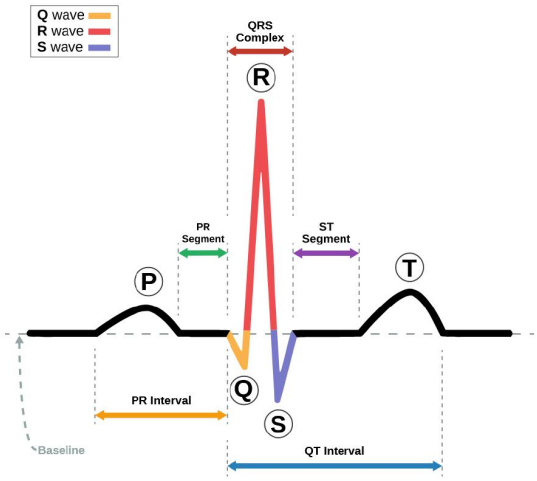
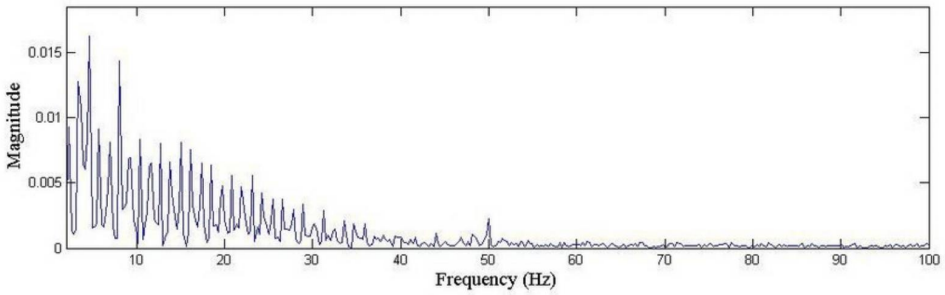


Figure 2.1: ECG waveform.

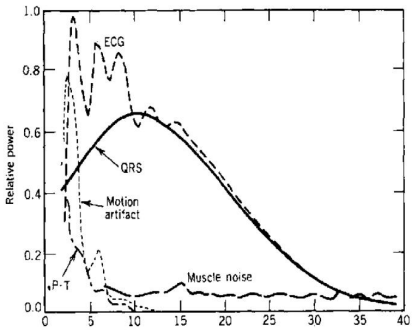
The U wave is hard to appreciate and typically is ignored.

In addition, there may occur variations or other contributions symptomatic of abnormal activity of the heart.

To better appreciate the information carried by the ECG signal, it should be converted by means of the Fourier transform, which transposes the time-domain signal into its frequency-domain representation, making it possible to determine the spectrum extension of the ECG signal.
The frequency range of the ECG signal typically extends from 0.05 Hz to 100 Hz, with most of the energy concentrated in the range of 0.5 Hz to 40 Hz.
The low-frequency components of the ECG signal (below 0.5 Hz) are associated with respiratory and other physiological movements, while the high-frequency components (above 40 Hz) are mostly noise and interference.



(a) Full frequency range[1].



(b) Useful frequency range.

Figure 2.2: Frequency spectrum of the ECG signal.

The ECG is normally displayed on a grid, with time along the horizontal axis and voltage on the vertical axis.

[1]At 50 Hz can be appreciated an extraneous spike due to the powerline interference.

The standard time reference represented on this grid is 1 second for 25mm (25mm/sec). The interpretation is as follows:

- The small box of 1 mm × 1 mm represents 0.1 mV × 0.04 seconds.
- The large box of 5 mm × 5 mm represents 0.5 mV × 0.20 seconds.

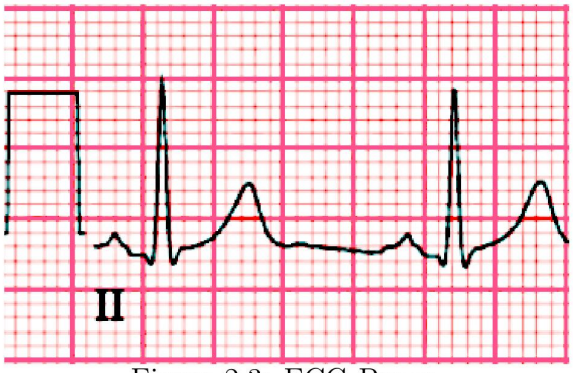


Figure 2.3: ECG Paper.

Scaling the information to the right scale on the graph is extremely important in identifying abnormal changes in the cardiac rhythm or other conditions such as ventricular hypertrophy, however multiple conditions can be identified by just observing the signal profile.
Typical variations in the profile associated with pathologies include the absence of the P wave, a sinusoidal signal pattern or even a "saw tooth" profile.

Compared to the classic representation of the ECG, which typically shows the variation of electric potential between right and left arms, different waveforms can be appreciated depending on the angle of observation of the cardiac activity.
These different results are obtained by placing electrodes on different points of the body.
The complete electrocardiogram analysis is the so called 12-lead ECG.
In this setup, the patient lies on a bed, and ten adhesive electrodes are placed on the patient's limbs and on its chest. The changes of the heart's electrical potential are then measured from twelve different angles and recorded for a certain period of time, usually ten seconds.
In this way, a complete overview of the heart's activity is captured during each phase of the cardiac cycle.

2.1.1 Limb & Precordial Leads

The difference in the electrical potential between two points of the human body represents a derivation.
Using 2 electrodes, only a single derivation can be obtained, depicting a single ECG waveform. In the medical field, a derivation is also called lead.
As anticipated, combining up to 10 electrodes it is possible to obtain 12 different derivations, that are subdivided into two main groups, these are the limb leads and the precordial leads.
The main difference in between this two categories concerns the angle of observation with which they inspect the heart: dividing the human body into two planes, as shown in fig.2.4, limb leads observe cardiac activity in the vertical (frontal) plane of the body while precordial leads, also known as chest leads, are located anteriorly on the chest and are used to detect signals traveling in the horizontal (transverse) plane, perpendicular to the previous one.

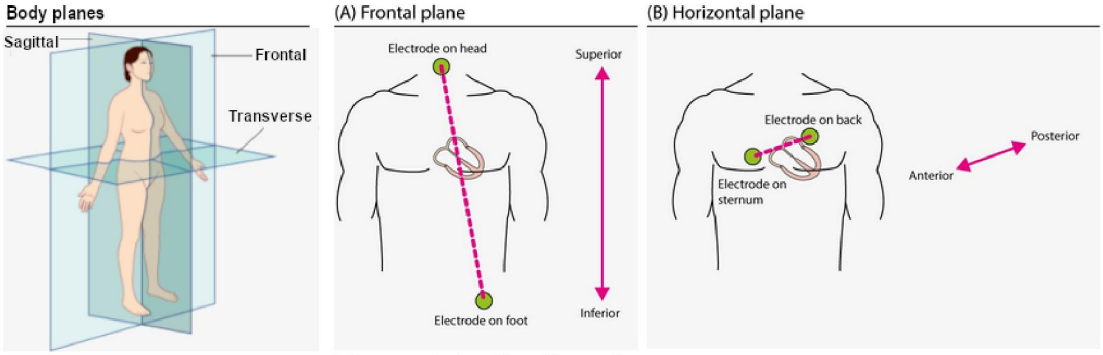


Figure 2.4: Cardiac planes.

While the limb leads provide a general overview of the cardiac condition, the precordial ones (V1, V2, V3, V4, V5 and V6) are particularly useful for assessing the status of the left ventricle, which is responsible for pumping oxygenated blood to the body.
In particular, the latter are useful for detecting changes in the ST segment and the T wave, which are indicative of a variety of cardiac conditions, including myocardial infarction, left ventricular hypertrophy and cardiomyopathies.
With regard to limb leads, however, it is necessary to make a further distinction between Einthoven's and Goldberger's leads.

2.1.2 Einthoven and Goldberger's leads

The limb leads consist of six different signals, namely I, II, III, aVF, aVR and aVL, located in the frontal plane.
The first three leads, respectively leads I, II and III are the Einthoven's original leads, and they can be displayed using the Einthoven's triangle.
The remaining are the aVR, aVL and aVF leads, first defined by the famous American cardiologist Lewis Goldberger; they are called unipolar leads because they refer to the average value between two electrodes.
The letter "a" stands for augmented, "V" for voltage and "R" is right arm, "L" is left arm and "F" is foot.
Usually, lead aVR is inverted into lead -aVR, since it is recommended as it may facilitate interpretation.

Leads I, II and III compare electrical potential differences between two electrodes.
Lead I compares the electrode on the left arm with the electrode on the right arm, of which the former is at the lower potential, observing with an angle of 0°.
Lead II compares the left leg with the right arm, with the leg electrode being at the higher potential. Therefore, lead II observes the heart from an angle of 60°.
Lead III compares the left leg with the left arm, with the leg electrode being the exploring one; it observes the heart from an angle of 120°.
The spatial organization of these leads forms a triangle in the chest knonw as the Einthoven's triangle.

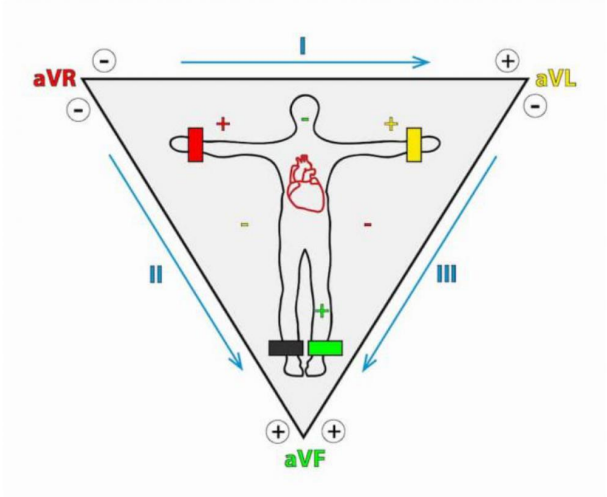
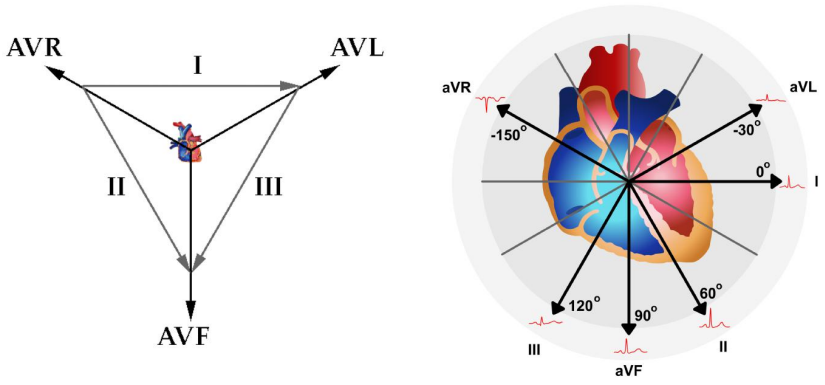


Figure 2.5: Einthoven's triangle.

For a better understanding of the relationships between the Einthoven and Goldberger's leads, all six limb leads are often presented in a circular Hexaxial coordinate system, as reported in figure 2.6b.
In this representation system, it is assumed that, according to the Einthoven triangle, Lead I defines 0° in the frontal plane. This also means that lead I "views" the heart from an angle of 0° and, according to what was aforementioned, the same principles apply to lead II and lead III.
However, it should be noted that, in this model, the reference is provided by the Goldberger's central terminal, conceptually localized at the center of the heart.
Looking at the representation, shown in fig.2.6a, can be easily defined the Goldberger's leads: they represent vectors starting from the barycentre of Einthoven's triangle and are bisectors of its angles, even if with greater amplitude.
By dividing the angles of the triangle formed by leads I, II, III into two equal parts, it results that the six leads allow to divide the front plane into areas of 30° each.
At this point it is evident that there is a relationship between the different limb leads.



(a) Limb vectors. (b) Hexaxial coordinate system.
Figure 2.6: Circular Hexaxial coordinate system representation.

2.2 Implementation of the theory

The aim of this analysis is to create a portable device for the acquisition of the six-lead ECG, for this reason the study will develop around the mathematical descriptions of limb leads, while the study of the precordial leads is out of the scope of this project.

Unfortunately, a variety of noise sources can impact ECG recordings, interfering with the reliability of the interpretation of the ECG signal and, in order to provide accurate cardiac disease diagnosis and therapy, it is essential to understand the typical causes of ECG noise.
There are mainly four types of artifacts encountered in ECG signals: baseline wander, powerline interference, electromyography noise and electrode motion artifacts. In the following, for the purpose of understand how to eliminate all of these undesirable contributions, they will be briefly discussed.

2.3.1 Baseline Wander (BW)

Baseline wander is the effect where the base reference seems to oscillate with a very slow period, moving up and down rather than be straight. This causes the entire signal to shift from its normal base.
In ECG signal, the baseline wander is caused due to improper electrodes (electrode-skin impedance), patient's movement and breathing (respiration).
Figure 2.7 shows a typical ECG signal affected by baseline wander.

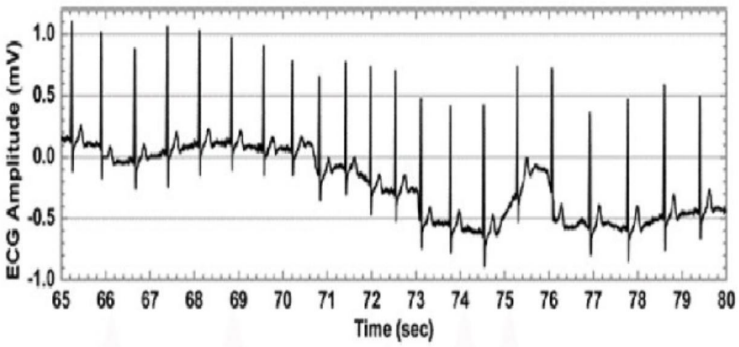


Figure 2.7: ECG Signal with baseline wander

The frequency content of the baseline wander is normally in the range of 0.5 Hz, however, increases with greater body movement during exercise or stress tests. Since the baseline signal is a low frequency signal, Finite Impulse Response (FIR) and high-pass filter with a cut-off frequency of 0.5 Hz can be used to estimate and remove the baseline in the ECG signal.

2.3.2 Powerline interference (PLI)

Electromagnetic fields caused by a powerline represent a common noise source in the ECG, as well as to any other electrical signal recorded from the body surface. Such noise is characterized by 50 or 60 Hz sinusoidal interference, possibly accompanied by a number of harmonics as shown in fig.2.8.
It is necessary to remove powerline interference from ECG signals as it completely superimposes the low frequency ECG waves, especially the P wave and T wave.

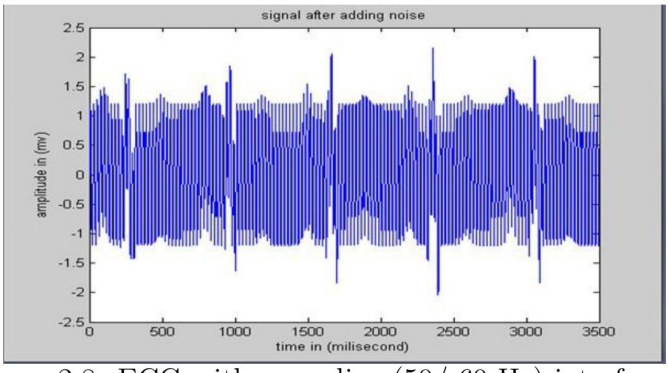


Figure 2.8: ECG with powerline (50/ 60 Hz) interference

A very simple approach to the reduction of powerline interference is to implement a notch filter defined by a complex-conjugated pair of zeros that lie on the unit circle at the interfering frequency.
The practical implications of using this type of filter are a high rejection of disturbances in a very selective way but also the introduction of ringing artifact in the output signal, due to the filter response, visible after the transient.
This is a source of further distortion of the signal as it introduces a contribution that overlaps the QRS complex and the T wave of the ECG signal.
For this reason, on the practical side, it was preferable to opt for a cascade of low-pass filters with a cut-off frequency below 50 Hz.

2.3.3 Electromyography noise (EMG)

Muscle noise is a significant issue in many ECG applications, particularly in recordings made during activity, where low amplitude waveforms may entirely disappear. Muscle noise is not eliminated by narrowband filtering, unlike baseline wander and powerline interference, indeed it provides a far more challenging filtering challenge since the spectral content of muscle activity significantly overlaps that of the

PQRST complex.
Numerical techniques can be utilized to lessen muscle noise by taking advantage of the fact that the ECG is a repeating signal, however the amount of effective noise reduction is insufficient.
Hence, there is still a need to develop signal processing techniques which can reduce the influence of muscle noise effectively.

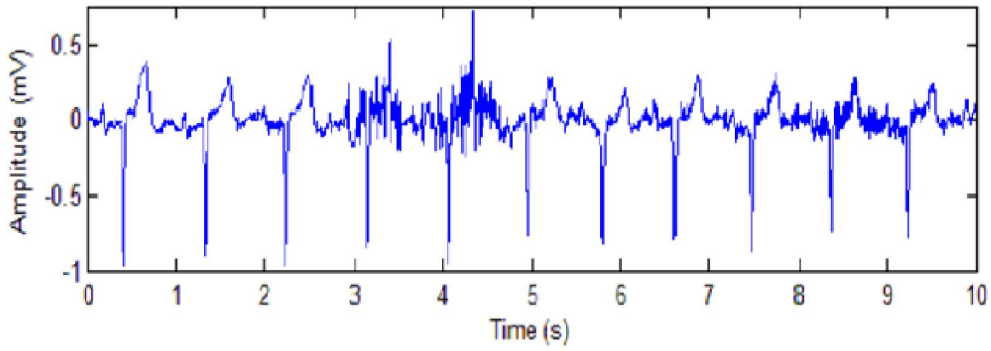


Figure 2.9: Electromyographic (EMG) noise

2.3.4 Electrode motion artifacts

Electrode motion artifacts are mainly caused by skin stretching which alters the impedance of the skin around the electrode.
The spectral content of motion artifacts significantly overlaps that of the PQRST complex, which makes them particularly difficult to eliminate.
They show as large-amplitude waveforms and are most prominent in the 1 to 10 Hz frequency range.
The following is an example:

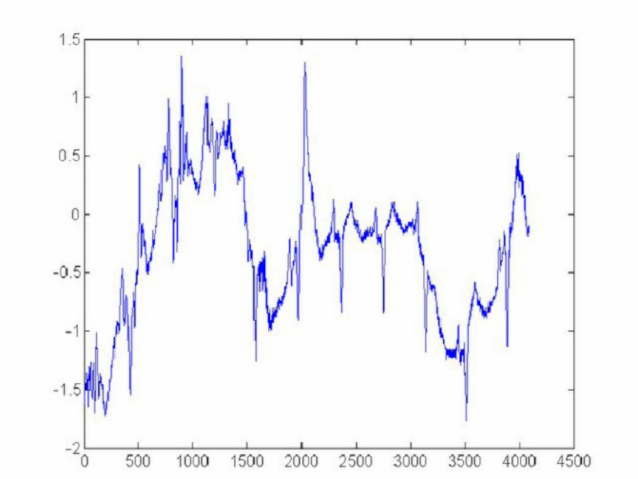


Figure 2.10: ECG affected by electrode motion artifacts

Chapter 3

Analog Signal conditioning

In order to acquire and appreciate the limb leads, it was necessary to develop a front end that would deal with the filtering and amplification of these signals, in order to eliminate the sources of noise and improve the quantization operation.

The search for a front end that would help clean up the ECG signal without being too aggressive was crucial.
At first, a front end was developed that would allow different filter configurations to be tested.
The basic idea was to implement a modular system of independently selectable filters to evaluate different performances.
This first solution is designed to work with a single lead, so taking a signal differentially between two electrodes.
The signal thus acquired is supplied to an instrumentation amplifier.
The instrumentation amplifier makes it possible to greatly reduce any unintended common-mode contributions, so that disturbances acting on both input connections are greatly attenuated.
In addition, the high input impedance prevents the unknown impedance of the electrode from creating a signal partition, attenuating it.
The instrumentation amplifier uses the voltage supplied by an integrator as a reference. This realize a high-pass filter with a cutoff frequency around 0.5Hz to eliminate the noise introduced by the baseline wander.
The signal thus acquired is passed to the network of interchangeable low-pass filters.
Among the different filters that can be selected are two first-order RC filters and two second-order Sallen-Key configurations.
These filters can be bypassed if required.
The purpose of this low-pass filter network is to research and find an alternative to using a notch filter to eliminate noise introduced by 50Hz power supply noise.

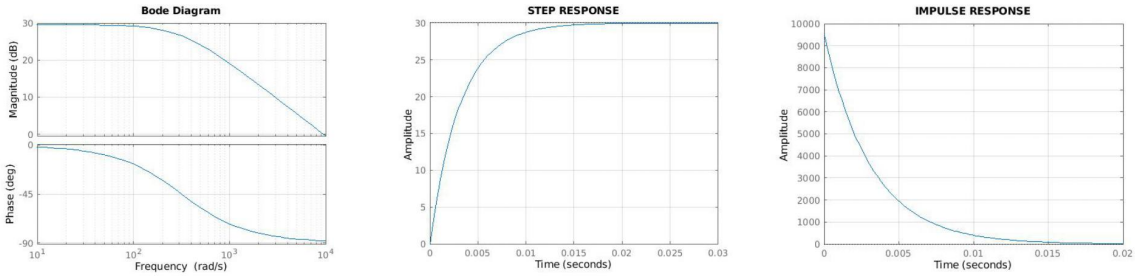


Figure 3.14: Output stage Bode diagram, step response and impulse response.

3.2 Preliminary results

The design of the final device went through the prototyping of the modular front end.
In this way it was possible to test the different filter configurations by simply soldering 0 ohm resistors to create different connections between the filters.
The developed device consisted of a front end and a 24-bit delta sigma ADC for the digitization of the filtered signal; the samples acquired were then sent to a development board via SPI interface and from the latter to a computer via UART to visualize the signal.
Figure 3.15 shows the modular front end testing board.

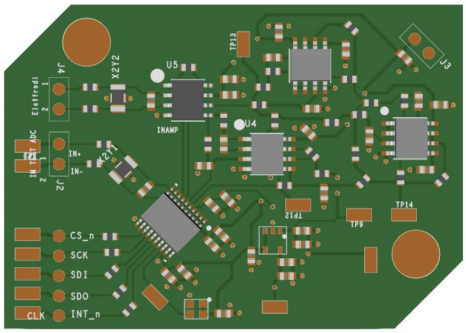


Figure 3.15: Front end testing board.

In addition, here below are briefly reported the results obtained with six different configurations of low pass filters, acquiring the ECG signals always in the same

This configuration showed good results without the use of additional digital filtering, almost comparable to those obtained with a professional electrocardiograph.

3.3.1 Full front end transfer function

Having chosen the filter configuration that exhibited the best relationship between noise rejection and distortion introduced, it is possible to derive the transfer function of the complete circuit, given by the product of the functions of the individual blocks studied above, which can be expressed as:

$$H = H_{in} \cdot H_{RCs} \cdot H_{out} \tag{3.20}$$

The theoretical results of the circuit response calculated with MATLAB and the experimental results are given below, where is reported a comparison between the ECG signal and those acquired simultaneously with a medical electrocardiograph, with and without additional digital filtering.

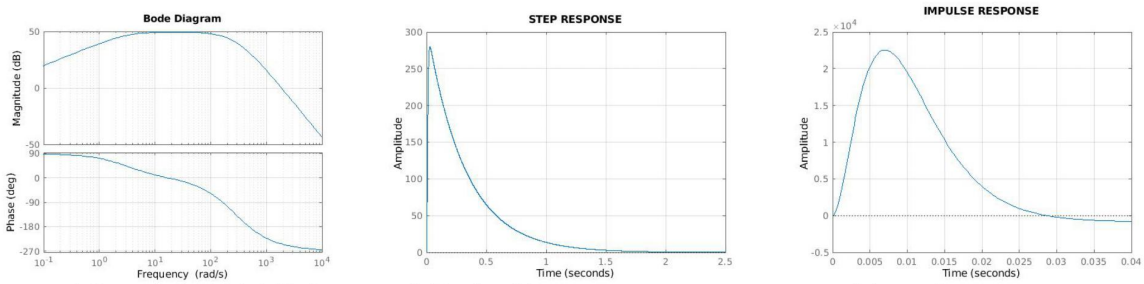


Figure 3.18: Full front end Bode diagram, step response and impulse response.

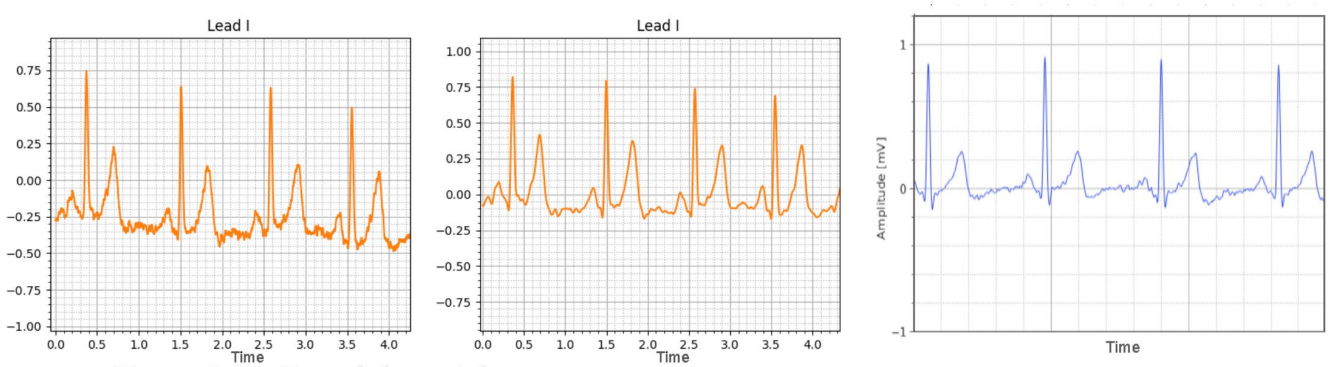


Figure 3.19: From left to right:

- Front-end acquired signal - no digital filtering.
- Front-end acquired signal - digital filtering.
- Professional electrocardiograph's signal.

Chapter 4

Board design

The aim of the project is to acquire a 6-lead electrocardiogram by means of a low-power, ultra small portable wireless device.
For this reason, having devised several possible variants, in the end it was decided to make a wearable device shaped like a wrist watch.
In this way, we chose to place an electrode under the device, as a contact on the left wrist and a second one above the device, to place the right index finger on it. The third electrode is placed on the strap, so you can rest it comfortably on your left ankle.

4.1 Device overview

The development took place, first of all, through the definition of the specifications that need to be outlined and respect.
Here are reported the features and the requirements of the device:

- Deliver a 6-lead ECG;
- Ultra small design (Radius = 2 cm, Height = 0.6 cm);
- Portable and Wearable on the wrist;
- Rechargeable through separate type-C USB charger;
- Low energy Bluetooth 5.3;
- Low power;
- Four layer PCB;

- Double layer SMT;

Another design phase that evolved during the development of the thesis was the design of the device itself.
Below are reported the drawings of the initial sketches of the project and of the main functional blocks, with the aim of providing the reader with a concrete vision of the product development process.
Furthermore, the drawings of the main functional blocks show how the product is structured to implement the specific functionalities required, providing a visual understanding of the key blocks of the product and their interconnections, helping the reader understand how the product works and how the different parts of the project have been integrated.

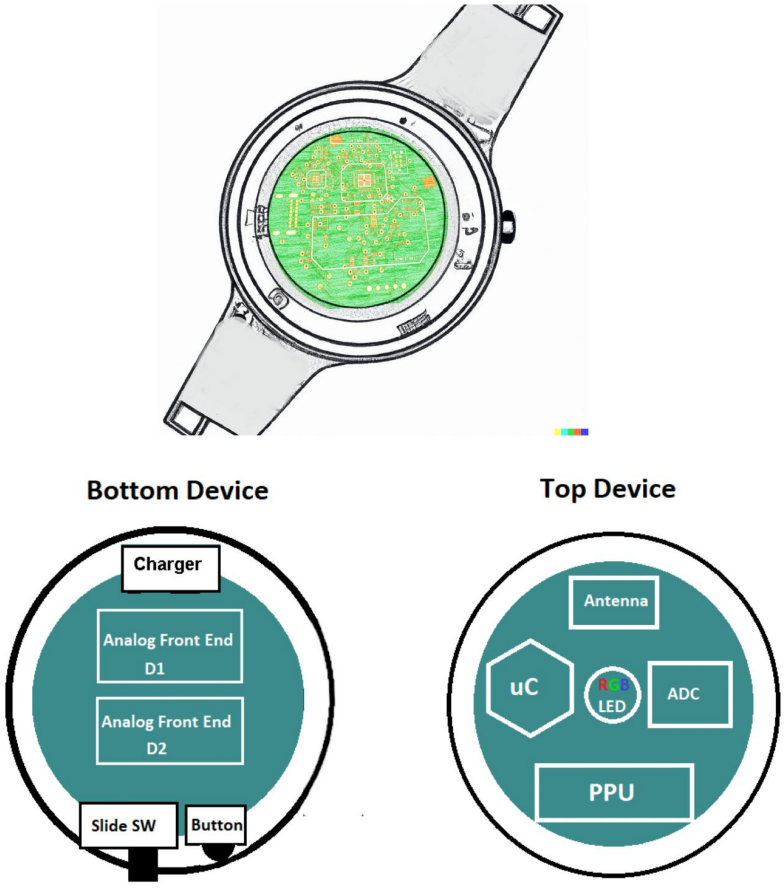


Figure 4.1: Board Sketch and Design.

4.1.1 Block Diagram

In order to further delineate the the functioning of the device, it is useful to refer to a graphical representation of the different functional units that are part of the appliance.
Each block represents a specific function or hardware unit and the diagram shows how these units are interconnected.
In figure 4.2 it is reported the block diagram of the device. It is possible to identify the five main functional blocks of the board:

- **Analog front end**. By means of the three electrodes, it acquires the ECG signals in a differential way. Common mode noise is reduced using an instrumentation amplifier. Signals are then filtered through a high pass filter at 0.5Hz and a low pass filter with a cut-off frequency of 28Hz. Finally, signals are amplified to optimize the SNR due to quantization error.
- **ADC section**. It acquires and discretizes the analog signals providing digital data to the Microprocessor. It requires an 8 MHz clock in order to work with a resolution of 24 bits.
- **Microcontroller**. It communicates using the SPI and a sireal wires JTAG interface. Processed data is sent to the smartphone through the Bluetooth antenna. It also manages all the control signal towards the different functional blocks.
- **Peripherals**. They represent the interface between the user and the hardware circuit.
- **Power Management**. It provides the supply voltage to all the system, as well as the bias and reference voltages. It manages the activation and shutdown of integrated circuits to optimize power saving; it turns off the device during charging, as required by law.

4.2.3 Microcontroller

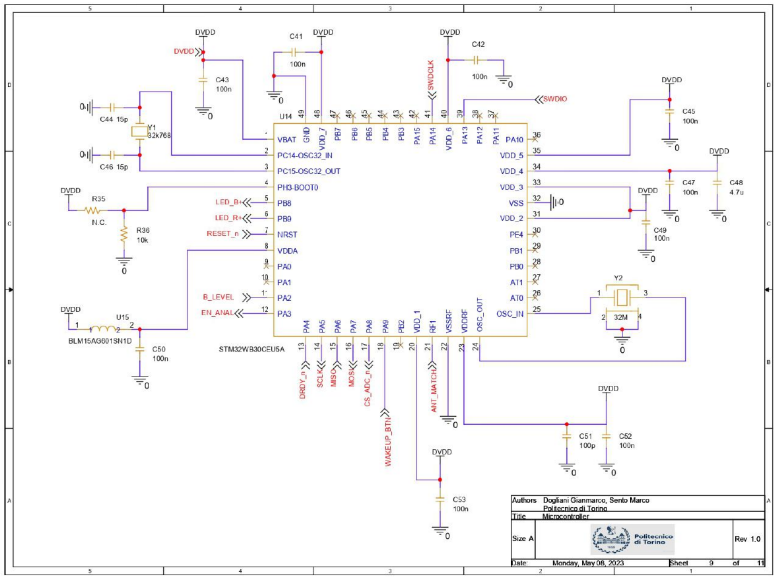


Figure 4.6: Microcontroller Section.

The ANT pin is connected to the chip antenna in order to send data from the internal Bluetooth module to the antenna.
P0.28 to P0.31 and P0.02 are used to communicate through the SPI interface with the ADC.
The pin P0.05, P0.11 and P1.09 communicate with the battery level sensor thanks to the I2C.
The pins P0.15 and P0.17 are used to change the duty cycle of the LEDs and so reduce their power consumption according to the requirement.
P0.03 is used in order to execute an interrupt and exit from low power states.
P0.10 and P0.09 are used to enable or disable integrated circuits when not useful (like the high power consumption 8MHz clock of the ADC).
SWDIO, SWDCLK and RESET$_n$ are used to connect the microcontroller to the JTAG connector.
Different capacitors are used to decouple the pins of the microcontroller.
Two external clock sources are used: in detail, a 32MHz and a 32.768 kHz crystals are required.

4.2.4 Peripherals

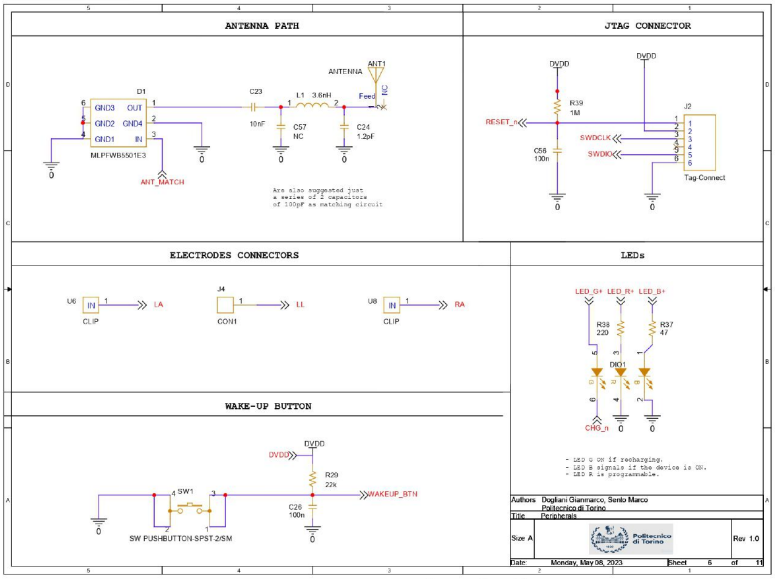


Figure 4.7: Peripherals Section.

In this last section all the communication elements are included:

- USB type C connector to recharge the battery;
- JTAG connector to program and debug the circuit;
- Different electrodes connectors (in order to thest the best solution);
- Push Button to wake-up the microcontroller from low power states;
- three LEDs.

The LEDs have been evaluated with the following formula:

$$R_{LED} = \frac{V_{max} - V_{nominal}}{I_{nominal}}$$

the relation with the discharge curve of the LiPo battery;
- Reference generator to provide stable 2.5V and 1.25V to the analog sub-part;
- High Side MOS to disable the analog sub-part in order to save energy.
- Green LED APTD1608LCGCK.

The green Led is connected between the voltage provided by the USB charger and the CHG pin of MAX1555 to signal the user that the battery is charging.

4.3 Components selection

4.3.1 Microcontroller

The µC is the core of the system: it must collect the data acquired by the ADC through the SPI interface, it receives the battery status from the I2C connection and must communicate through the Bluetooth antenna with the Smartphone.
The controller is also in charge of managing the power consumption, going into sleep mode and activating or deactivating entire portions of the system.
It communicate with the users also through light indicators (LEDs), push buttons and the JTAG interface (for programming).
The project was initially designed around the nRF52833 µC but it has changed in favour of the STM32WB30RG, since it offers the same features but with a more user friendly and flexible IDE that allows also the integration in the firmware of ANN developed with TensorFlow.

- **STM32WB55RG**

The STM32WB55xx and STM32WB35xx multiprotocol wireless and ultra-low-power devices embed a powerful and ultra-low-power radio compliant with the Bluetooth® Low Energy specification 5.3.
The devices are designed to be extremely low-power and are based on the high-performance Arm® Cortex®-M4 32-bit RISC core operating at a frequency of up to 64 MHz.
The devices embed high-speed memories (up to 1 Mbyte of flash memory for STM32WB55xx, up to 512 Kbytes for STM32WB35xx, up to 256 Kbytes of SRAM for STM32WB55xx, 96 Kbytes for STM32WB35xx.
Direct data transfer between memory and peripherals and from memory to memory is supported by fourteen DMA channels.
It offers an overall lower power consumption in respect to the nRF52833 SoC and a more user friendly and flexible IDE that allows also the integration in the firmware of ANN developed with TensorFlow.

The drawback is a slightly larger occupied area on the PCB in respect to the nRF52833.

Main features

- 64 MHz Arm Cortex-M4
- 1 MB Flash, 256 KB RAM
- 6 different Low Power Modes
- Bluetooth Low Energy 5.3
- 32MHz SPI, I2C, UART, DMA, NFC
- 12-bit ADC
- 6x timers (1x RTC, 3x General Purpose, 2x Low Power)
- 7x7 mm QFN48 Package

- Supply Voltage: 1.7V to 3.6V
- Shutdown mode current: 13 nA
- Standby mode current: 600 nA
- Stop mode current: 2.1 µA
- Active mode current: up to 53 µA/MHz
- RF current: Rx 4.5 mA / Tx (@ 0 dBm) 5.2 mA
- Up to 64MHz from a 32 MHz external crystal oscillator
- External 32.768 kHz crystal oscillator

Pinout and Package Outline

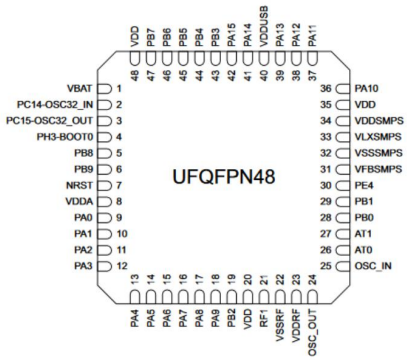


Figure 4.9: STM32WB55RG pinout and functions

Reference Layout

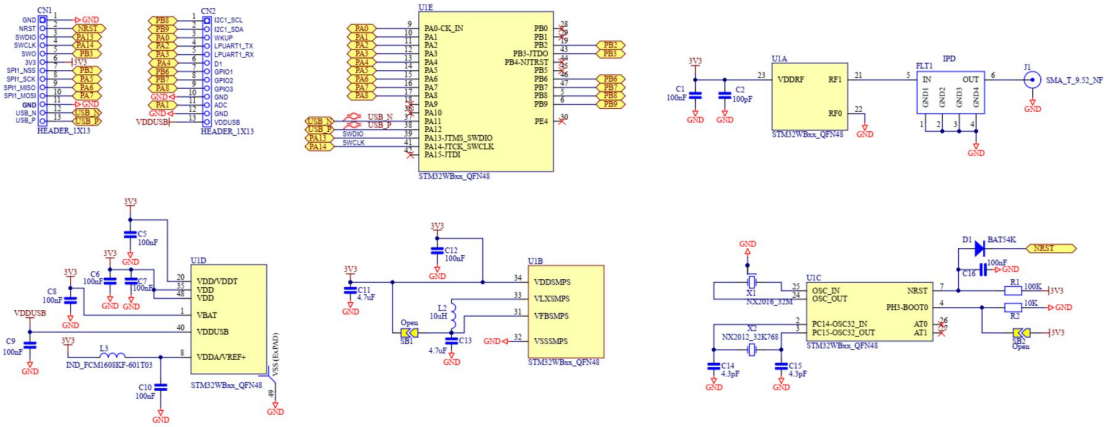


Figure 4.10: STM32WB55RG Reference layout

4.3.4 Peripherals

- **JTAG connector: TC2030**

The JTAG Connector is used to connect and program the microcontroller.
In order to maintain a compact design, the JTAG connector has been realized as six simple conductive contact pads.
It covers an area of only 5x5 mm^2.
The programmer cable is the so called PLUG-OF-NAILS cable.

Pinout and Package Outline

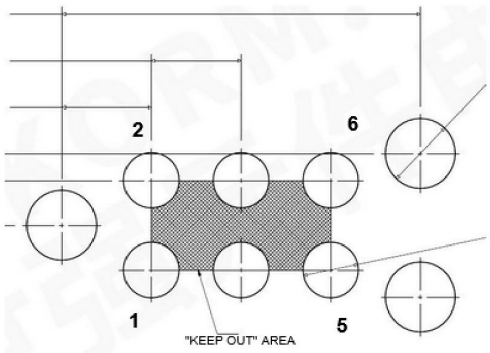


Figure 4.16: TC2030 pinout and functions

- **Antenna layout**

DA CAMBIARE

In order to avoid matching problems with more delicate devices such us chip antennas, it has been implemented a patch antenna for the Bluetooth still maintaining a small layout.

Main features

- Very small 0402 ceramic package
- Single feed

Electrical characteristics

- Operating Frequency: 2.4 - 2.5 GHz

- Impedance: 50 ohm

- Averange Gain: 1.95 dBi

Reference Layout

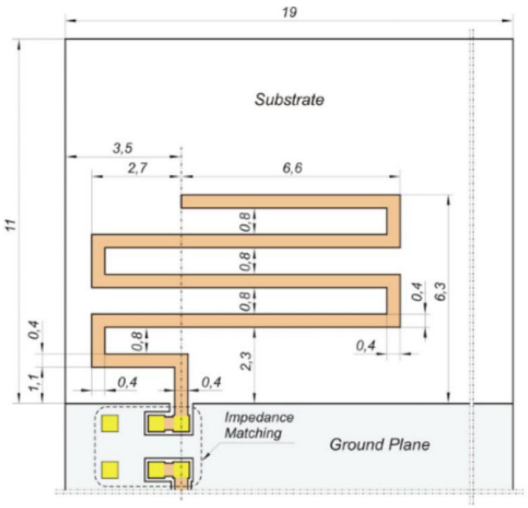


Figure 4.17: Antenna Reference layout

- **Push button: EVQPUK02K**

This push button is used in order to communicate with the microcontroller, it is specifically introduced in order to wake up the controller from the deep sleep mode.

Pinout and Package Outline

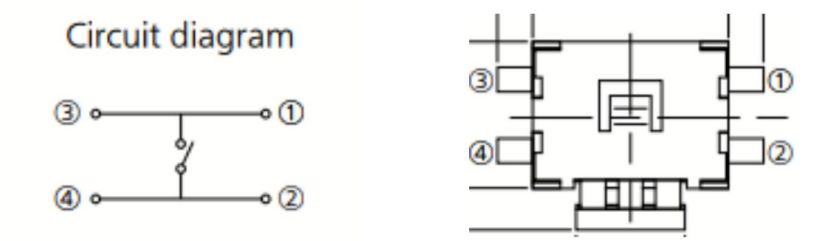


Figure 4.18: Push Button pinout and functions

- **RGB LED: EAST1616RGBA4**

The RGB LED is used to highlight the charge state and operation conditions of the µC.
The anodes are managed directly by the µC in order to control the power consumption by varying the duty cycle.
Since the RGB LEDs have different forward operating voltages and currents, it has been assumed as maximum continous current the lowest value of 6mA, then the series resistors have been choosen considering the different forward voltages of each colour and the different supply voltages applied to them.

- BLUE: is ON when the device is active
- GREEN: is ON when the battery is charging
- RED: is ON when the battery has low charge

Main features

- Small 6 pin package
- Individual anodes and cathodes

Electrical characteristics (T @ 25 $°C$)

- Max Forward Voltage: -R 2.4V -G 3.8V -B 3.8V
- Max Forward Current: -R 6mA -G 10mA -B 10mA

- Dominant Wavelength: -R 624nm -G 525nm -B470nm

Pinout and Package Outline

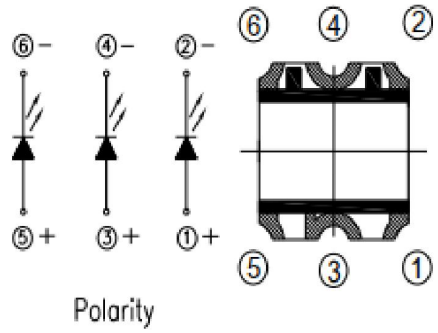


Figure 4.19: RGB LED pinout and functions

4.3.5 Power Management Section

- **LiPo battery: HPL402323-2c-190mAh**

A compact Li-Polymer Battery 3.7V 190mA for low power portable devices.

Electrical characteristics (T @ 25 $°C$)

- Connector: JST-SHR-02V-S
- Nominal voltage: 3.7V
- Voltage at end of discharge: 3.0V
- Charging voltage: 4.2±0.03V
- Capacity: 190 mAH
- Standard Charge 0.5CA
- Fast Charge 1CA
- Standard Discharge 0.2CA

Pinout and Package Outline

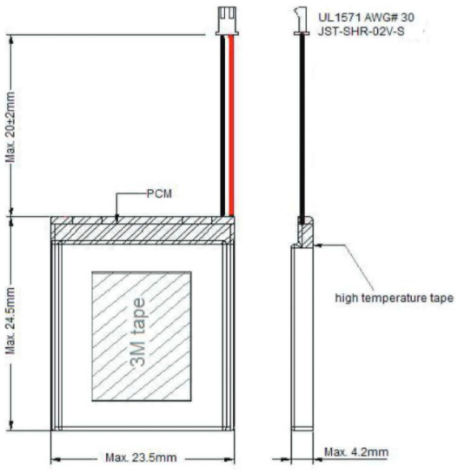


Figure 4.20: Battery pinout and functions

- **Battery Charger: MAX1555**

The Battery Charger is used to charge the single-cell LiPo battery from the USB source.
It operates with input voltages up to 7V and offers the CHG pin, used to detach the supply when in charge.
When CHG goes to a high-impedance state, the battery is fully charged and the charge current falls below 50mA.

Main features

- Charge from USB and/or AC Adapter
- Automatic Switch over when AC Adapter is plugged IN
- On-Chip Thermal Limiting Simplifies Board Design
- Charge Status Indicator
- 5-Pin Thin SOT23 Package

Electrical characteristics (T $\in$ [0,85]$°C$)

- DC Voltage Range: 3.7V to 7.0V
- USB Voltage Range: 3.7V to 6.0V
- DC to BAT Voltage Range: 0.1V to 6.0V
- BAT Regulation Voltage: 4.158V to 4.242V
- Max DC Supply Current: 3mA
- Max DC charging current: 340mA

Pinout and Package Outline

PIN	NAME	FUNCTION
1	USB	USB Port Charger Supply Input. USB draws up to 100mA to charge the battery. Decouple USB with a 1µF ceramic capacitor to GND.
2	GND	Ground
3	$\overline{POK}$	Power-OK Active-Low Open-Drain Charger Status Indicator. $\overline{POK}$ pulls low when either charger source is present (MAX1551 only).
	$\overline{CHG}$	Active-Low Open-Drain Charge Status Indicator. $\overline{CHG}$ pulls low when the battery is charging. $\overline{CHG}$ goes to a high-impedance state, indicating the battery is fully charged, when the charger is in voltage mode and charge current falls below 50mA. $\overline{CHG}$ is high impedance when both input sources are low (MAX1555 only).
4	DC	DC Charger Supply Input for an AC Adapter. DC draws 280mA to charge the battery. Decouple DC with a 1µF ceramic capacitor to GND.
5	BAT	Battery Connection. Decouple BAT with a 1µF ceramic capacitor to GND.

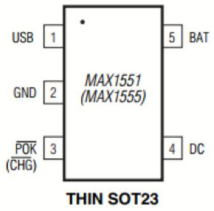


Figure 4.21: MAX1555 pinout and functions

Reference Layout

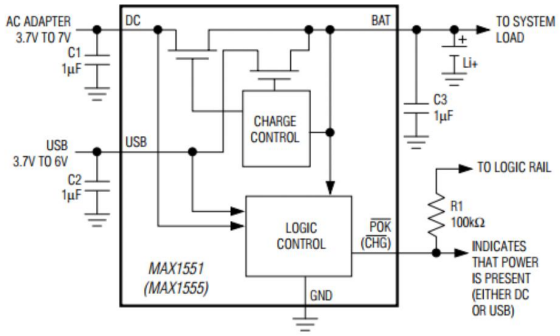


Figure 4.22: MAX1555 Reference layout

- **Voltage Regulator: MAX1759**

The Voltage Regulator is used to generate a regulated output voltage from a single cell LiPo battery.
It provides a fixed voltage (3.3V) to all the circuit, accepting in input a voltage up to 5.5V.
It's also provided with a shutdown pin, exploited for turn off the device during charging.

Main features

- Regulated Output Voltage (Fixed 3.3V or Adjustable 2.5V to 5.5V)
- Shutdown Mode
- Load Disconnected from Input in Shutdown
- Short-Circuit Protection and Thermal Shutdown
- Small 10-Pin µMAX Package

Electrical characteristics (T $\in$ [0,85]$°C$)

- Input Voltage Range: 1.6V to 5.5V
- Output Voltage: 3.17V to 3.43V
- Minimum Output Current: 100mA
- Quiescent Supply Current (Vin > 4V): 90µA
- Shutdown Supply Current: 5 µA
- Efficiency: 90%
- Switching Frequency: 1.2-1.8 MHz

Pinout and Package Outline

PIN	NAME	FUNCTION
1	POK	Open-Drain Power-OK Output. POK is high impedance when output voltage is in regulation. POK sinks current when V_{FB} falls below 1.1V. Connect a 10kΩ to 1MΩ pull-up resistor from POK to V_{OUT} for a logic signal. Ground POK or leave unconnected if not used. POK is high impedance in shutdown.
2	$\overline{SHDN}$	Shutdown Input. Drive high for normal operation; drive low for shutdown mode. OUT is high impedance in shutdown.
3, 4	IN	Input Supply. Connect both pins together and bypass to GND with a ceramic capacitor (see *Capacitor Selection* section).
5	GND	Ground. Connect GND to PGND with a short trace.
6	PGND	Power Ground. Charge-pump current flows through this pin.
7	CXN	Negative Terminal of the Charge-Pump Transfer Capacitor
8	CXP	Positive Terminal of the Charge-Pump Transfer Capacitor
9	OUT	Power Output. Bypass to GND with an output filter capactor.
10	FB	Dual-Mode Feedback. Connect FB to GND for 3.3V output. Connect to an external resistor divider to adjust the output voltage from 2.5V to 5.5V.

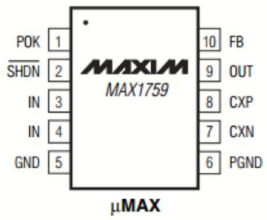


Figure 4.23: MAX1759 pinout and functions

Reference Layout

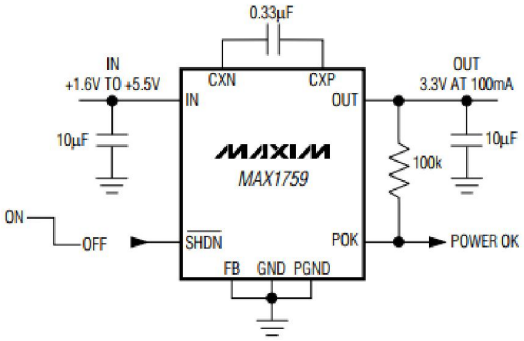


Figure 4.24: MAX1759 Reference layout

- **Battery Level Tracker:**

VEDERE SE METTERE qualcosa

Fuel Gauge device is used to track the battery relative state-of-charge (SOC) continuously over widely varying charge and discharge conditions.

- **Voltage Reference: REF2025**

This device is used in order to provide two different voltage references, of 1.25V and 2.5V, to the analog circuits.
Since it is used a unipolar supply voltage, it is useful to translate the signal around 1.25V.
It also provides the Reference Voltage for the ADC.
It offers also a shutdown pin.

Main features

- Two outputs, VREF and VREF/2, for convenient use in single-supply systems
- High initial accuracy: ±0.05
- VREF and VBIAS tracking overtemperature
- Microsize package: SOT23-5

Electrical characteristics (T @ 25 $°C$)

- Supply Voltage: -0.6V to 6V
- Low dropout voltage: 10 mV
- High output current: ±20 mA
- Low quiescent current: -Active mode: 360 µA -Shutdown mode: 5 µA

Pinout and Package Outline

PIN		I/O	DESCRIPTION
NO.	NAME		
1	V_{BIAS}	Output	Bias voltage output (V_{REF} / 2)
2	GND	—	Ground
3	EN	Input	Enable (EN ≥ V_{IN} – 0.7 V, device enabled)
4	V_{IN}	Input	Input supply voltage
5	V_{REF}	Output	Reference voltage output (V_{REF})

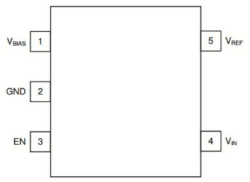


Figure 6-1. DDC Package SOT23-5 (Top View)

Figure 4.25: REF2025 pinout and functions

Reference Layout

4.3.6 Bill of Materials

Schematic parts have been characterized with custom properties in order to speed up BOM generation. Specifically, the following properties have been added:

- Description;
- Distributor and Distributor Part Number;
- Manufacturer and Manufacturer Part Number;
- Unit Price;

The resulting total price of the bill of material is 70€ including tax.

4.4 Printed Circuit Board design

The PCB is very small as the aim is to be positioned like a wristwatch.
Its final dimensions are in fact only 2 cm of radius.
Taking into account that the battery will have to be placed under the PCB, the final device will have a thickness of only 6mm.

The design is very dense with components, the analog part is very cumbersome for such a small space and probably in a future update a completely digital filtering will be preferred.
Using a double layer would not have been possible.
Using a four layer with a traditional stackup (ground and supply in between) would not have been possible as there is a high number of gnd VIA that hinder design quite a bit.
In addition, the analog part refers to a different net, AVDD.
For these reasons it was decided to make the PCB with the following stackup:

In addition, the analog part refers to a different net, AVDD.
For these reasons it was decided to make the PCB with the following stackup:

- Top Layer: Analog front end and power unit
- Layer 1: GND (digital)
- Layer 2: GND (analog)
- Bottom Layer: Digital part (µC, ADC and peripherals)

Note that in the final design the top layer will actually face down, housed above the battery.

The antenna has been put on the upper part of the circuit and as close as possible to the microcontroller to avoid parasitics.
Below it has not been put the Ground plane.
The microcontroller has been placed in the center of the circuit to facilitate connections to other components.

The number of traces in the GND layers has been kept to the minimum to avoid high current density points. The two GND are elettrically connected.
The charger connector and the buttons have been positioned on the borders of the circuit to make it easier to use.
A large housing has been dedicated in front of the LiPo connector to facilitate the connection.
In the following pages are shown:

- Top layer: full view - connections only
- Layer 1: GND
- Layer 2: GND
- Bottom Layer: full view - connections only
- PCB renders.

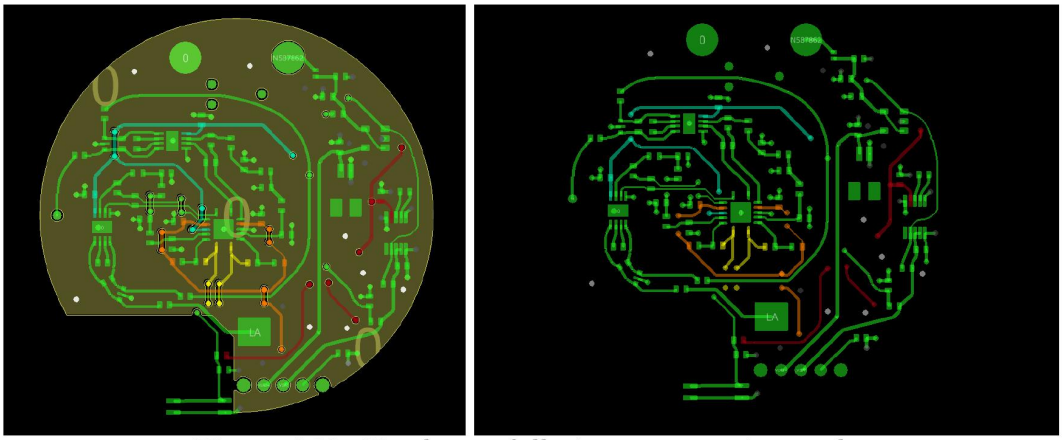


Figure 4.30: Top layer: full view - connections only.

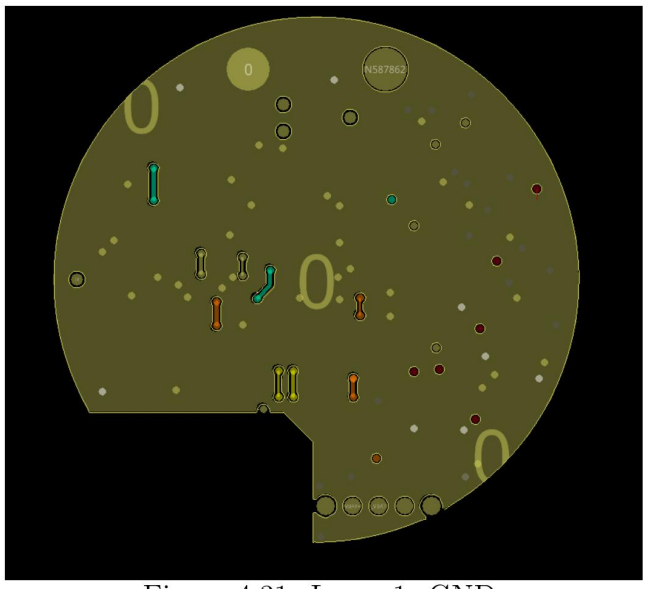


Figure 4.31: Layer 1: GND.

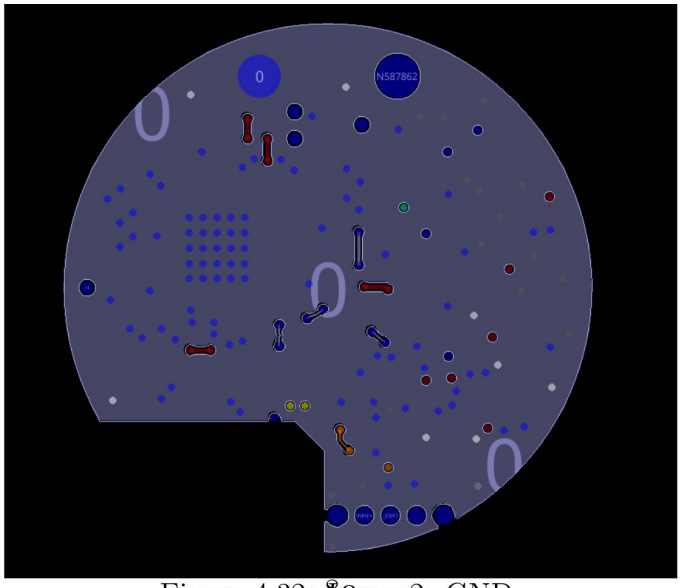


Figure 4.32: Layer 2: GND.

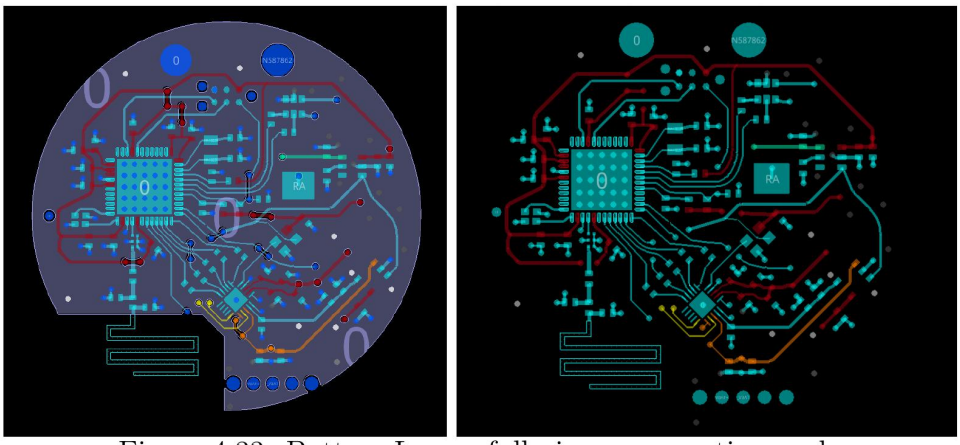


Figure 4.33: Bottom Layer: full view - connections only

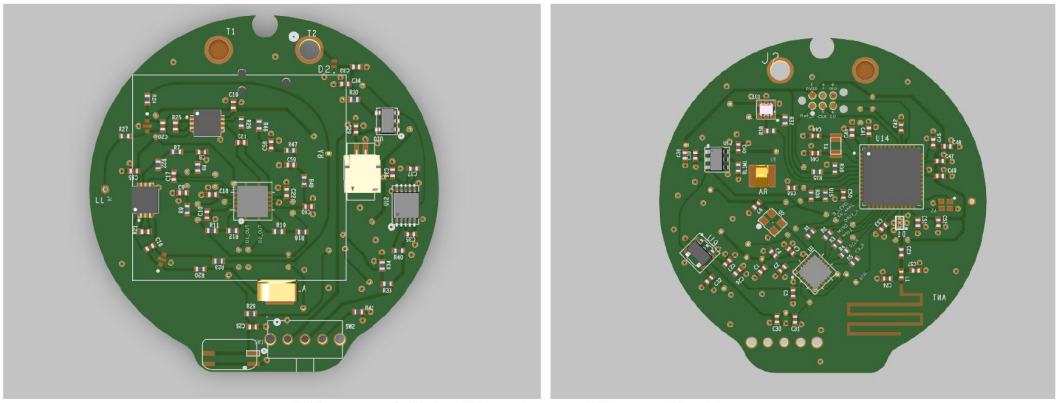


Figure 4.34: Renders: Top - Bottom

4.5 Mockup

This six-lead wearable device represents a breakthrough in ECG monitoring, revolutionizing the way cardiovascular health is assessed outside of traditional clinical settings. Its ergonomic design prioritizes user comfort and ease of use, ensuring optimal adherence and long-term wearability.
After the PCB design, the mock-up was realised: a wearable device resembling a watch, with an electrode placed in contact with the wrist, one on the upper shell and one under the strap, to be placed on the left leg for a complete six-lead ECG. Below is the final device made using Autodesk's fusion360 CAD and realised using the 3D printing technique.

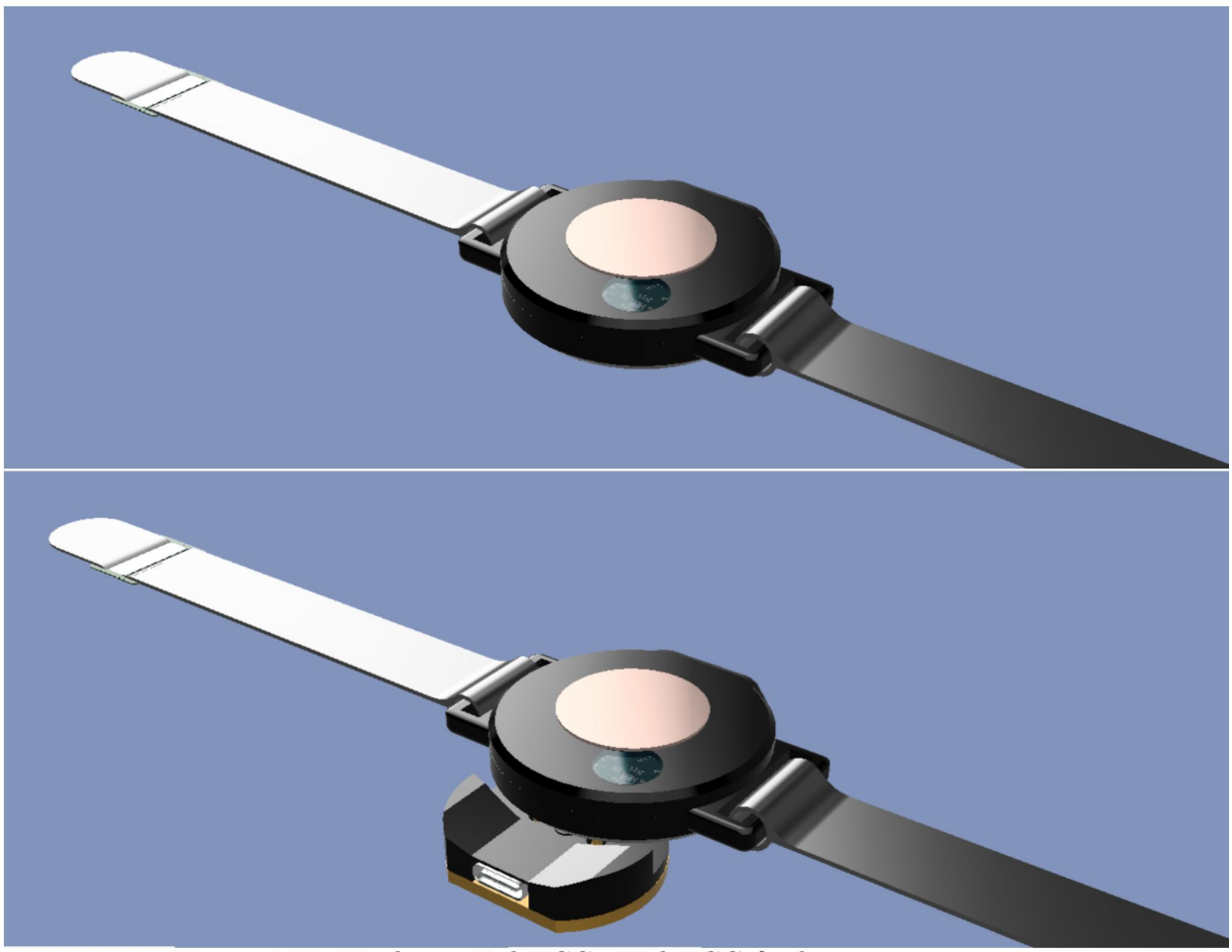


Figure 4.35: Mock-up: PulsECG - PulsECG & charger

Chapter 5

Firmware

This chapter focuses on the firmware development, written for the STM32WB30 microcontroller using the C programming language.
The primary objective of this firmware is to facilitate the acquisition of high-quality ECG signals from the human body and establish a reliable communication link with a smartphone. To accomplish this, the firmware must effectively manage the communication between the analog front end and the digital components of the device.
One of the key components in this wearable device is a 24-bit analog-to-digital converter (ADC), which is responsible for converting the analog ECG signal into a digital format. The communication between the ADC and the microcontroller is established using the Serial Peripheral Interface (SPI) protocol. The firmware must implement the necessary SPI functionalities to enable efficient data transfer between these components, ensuring accurate and reliable signal acquisition.
Furthermore, the firmware plays a critical role in establishing and managing the Bluetooth connection with the smartphone. This involves implementing the Bluetooth Low Energy (BLE) protocol stack, which enables energy-efficient communication over short distances. The firmware must handle tasks such as device discovery, connection establishment, and data transmission/reception, adhering to the relevant Bluetooth profiles and specifications.

The development of firmware for a Bluetooth medical wearable device presents various challenges, including limited resources such as memory and processing power, and it should support real-time data visualization for data storage and retrieval on the smartphone. This chapter aims to delve into the technical aspects of developing the firmware, emphasizing the integration of Bluetooth communication, management of the analog front end via SPI, and ensuring accurate and reliable acquisition of ECG signals.
It is through the developing of efficient firmware that it is possible to advancing the

field of wearable healthcare technology, ultimately improving patient monitoring and healthcare outcomes.

5.1 Code overview

Since the device is built around the STM32WB30 microcontroller from ST, a large part of the initialisation of peripherals and hardware is performed by the STMCubeIDE, an integrated development environment (IDE) provided by STMicroelectronics that offers a range of features and tools to simplify the initialization of hardware components.
It provides a graphical interface and a toolchain (compiler and linker) for the cross-compilation of the embedded system.
An IDE is usually used to write the code, compile, program the target and debug/run the application.
This development environment also offers a device configuration tool called MX that takes care of generating: the project (generates Makefile), the libraries and the hardware initialisation procedure.
The source files of the generated project are then available in the environment.
The resulting firwmware is generated for bare metal systems.
In these systems, the application is the only software on board. A series of start-up procedures are therefore linked together with the user application to initialise the hardware.
In the FLASH we have the interrupt vectors, the application code and the constants.
In the RAM we have the stack and the variables.
Since we do not have a bootloader, all hardware initialisation tasks must be performed by the application, which performs the initialisation of GPIO, SPI, Timer and ADC (of the uC); generating the code for handling interrupts and microcontroller resources as well as the configuration of the Bluetooth stack, by means of special APIs, enabling a fast management of Bluetooth connections, services and characteristics, and the exchange of data between the board and connected devices.

Figure 5.1: STMCubeMX configuration and peripherals.

For these reasons the chapter will focus more on the description of the functions and application-dependent modules, implemented in order to obtain an harmonious operation of the system's various sub-parts.
It is important to understand from the beginning that the user code is managed via a scheduler called SEQ (sequencer). It takes care of executing the tasks set by the user in sequence according to different priorities.

The main modules, that make up the firmware, deal with 3 main aspects:
1) Management of core operations and events
2) Management of the ADC via SPI
3) Bluetooth management

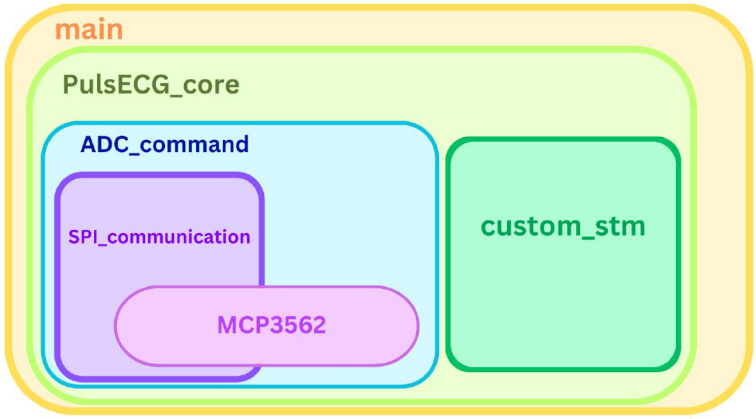


Figure 5.2: Modules hierarchy.

5.2 Application core

Following the enabling of the peripherals by the STMCubeIDE development tool, as mentioned earlier, their initialisation takes place in the **main.c** file. Here, it is possible to display the configurations for each of them, such as the SPI, GPIO and Real Time Clock.

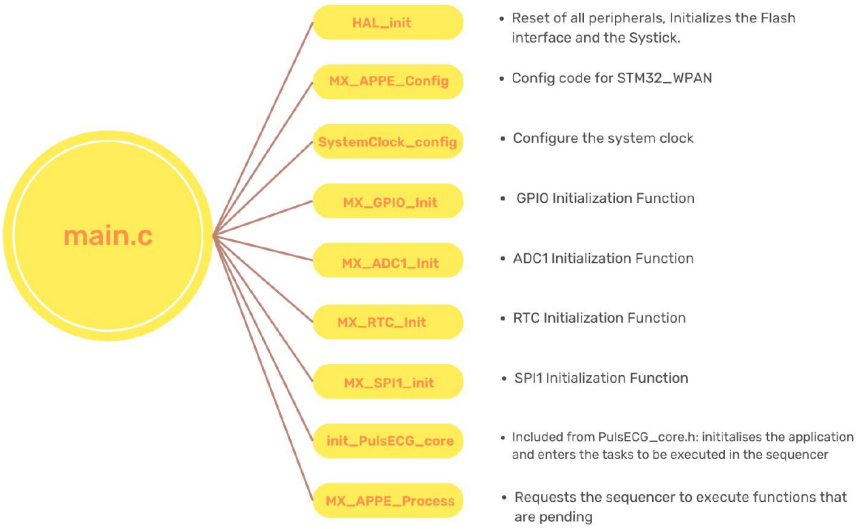


Figure 5.3: main file functions.

At power-up, the firmware takes care of initialising the hardware, and basic functionality is set up: variables are initialised and tasks that will be executed are fed into the scheduler.
It is at this stage that the function *init_PulsECG_core* takes care of initialising the application that will be executed on the device.

The **PulsECG_core** module contains the body of the application. This module takes care of orchestrating the operation of the device. There are basically two tasks to be performed: monitoring the battery charge and starting and stopping the acquisition of ECG signals.
Upon receiving an operation, such as a write or read, from the central device via Bluetooth, the module enables the sequencer to perform the correct function.
If the request is for a battery read, the sequencer will execute the function *send_BLevel* once, which causes the value corresponding to the battery charge percentage to be sent via Bluetooth.
This value is independently and periodically updated by initiating a battery voltage sampling at set intervals, as defined in the function, *update_BLevel*, via the microcontroller's internal ADC.
If the request is to read the ECG signal then a sequence of functions is executed: *start_stop_acquisition* receives as input the time frame for which the acquisition lasts, enables the analogue power supply and clock of the ADC, adjusts the status LED and start acquisition of ECG signals by sending the appropriate opcode to the ADC. This causes the functions *acquire* and *sendSample* to be iteratively re-entered into the scheduler for the duration of the acquisition.
As the name suggests, *acquire* receives the converted sample from the ADC via SPI while *sendSample* sends the samples almost in real time to the Central device.
Functions requiring execution by the sequencer must be declared in the file **app_conf.h**.
In this way, an ID can be defined that maps the function in the scheduler.
There are two lists of IDs, one for tasks that require bluetooth and one for those that do not communicate with it.
This allows the API provided by the scheduler to be used for inserting or removing functions from the list of tasks to be executed.

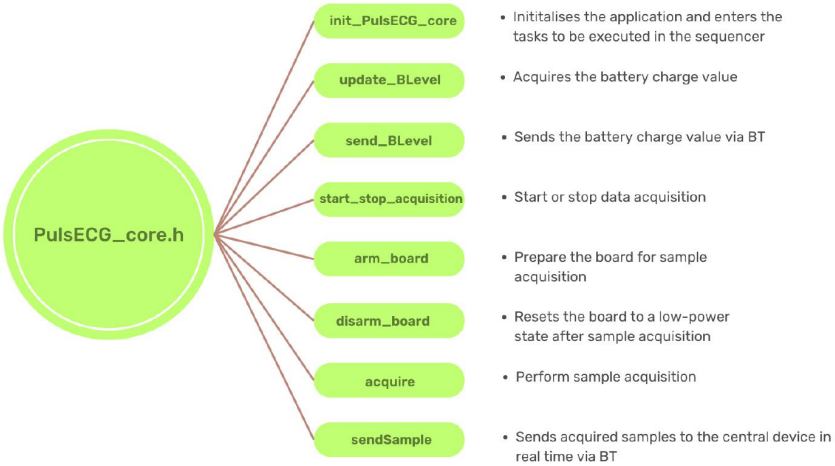


Figure 5.4: PulsECG_core file functions.

It is clear that this module handles the operations that are performed by the device, but in order to obtain the desired operations from the peripherals, it is necessary to format the requests correctly. This is why the different operations pass through different modules: **SPI_communication**, **ADC_command**, **MCP3562** and **custom_stm**.

5.3 SPI interface

As seen, In embedded systems is crucial to be able to manage communication between components, and of all of them, the interface that play a vital role in facilitating data transfer is the Serial Peripheral Interface (SPI).
It is widely adopted due to its simplicity, versatility, and high-speed capabilities. It allows for the exchange of data between a master device (typically a microcontroller or microprocessor) and one or more slave devices (peripheral devices such as sensors, displays, ADCs, DACs, memory chips, etc.).
SPI operates on a master-slave architecture, where the master device initiates and controls the communication. The master device generates clock pulses and selects the slave device with which it wishes to communicate. The data transfer occurs simultaneously in both directions (full-duplex), meaning that the master can send data to the slave while receiving data from it, or vice versa.

The SPI interface consists of four main lines:

SCLK (Serial Clock): This line provides the clock signal generated by the master device. The clock rate is configurable and determines the speed of data transfer.

MOSI (Master Out Slave In): This line is used by the master to send data to the selected slave device. It carries the data bits serially, with each bit synchronized to the clock signal.

MISO (Master In Slave Out): This line is used by the master to receive data from the slave device. The slave sends data bits serially on this line, also synchronized to the clock signal.

SS/CS (Slave Select/Chip Select): This line is used by the master to select a specific slave device with which it wants to communicate. By activating the SS/CS line for a particular slave, the master establishes a communication channel with that device while deactivating other slave devices.

SPI supports different configurations and settings, which can be tailored according to the specific requirements of the embedded system. Some of the commonly configurable parameters include:

Clock Polarity (CPOL): CPOL determines the idle state of the clock line. It can be set to either high or low, indicating whether the clock is idle when high or low, respectively.

Clock Phase (CPHA): CPHA determines the timing of data sampling and shifting. It can be set to either the first or second edge of the clock cycle.

Bit Order: SPI supports two modes of data transmission: MSB (Most Significant Bit) first or LSB (Least Significant Bit) first. The selection depends on the specific device's data format.

Data Frame Size: The frame size specifies the number of bits in each data frame. Usually are transmitted 8 bits but it depends on the capabilities of the devices involved.

In the following are reported the settings used for the the SPI and and the conventions to be observed during the communication:

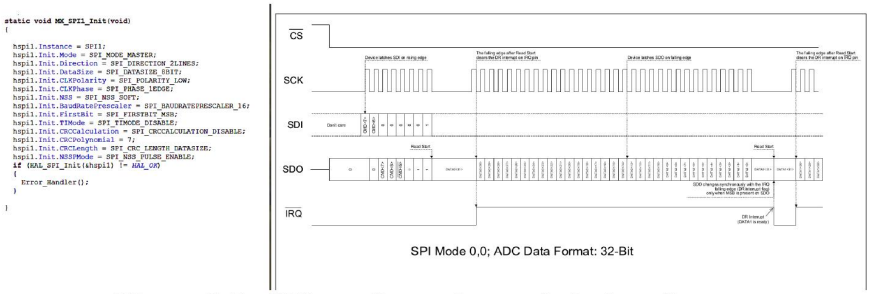


Figure 5.5: SPI configuration and timing diagram.

The module in charge of managing the communication via SPI with the 24-bit ADC is the **SPI_communication** module. It implements the functions needed to send bit sequences via SPI, read the bits received as a response and acknowledge and, if necessary, convert them into a format understandable to the developer.

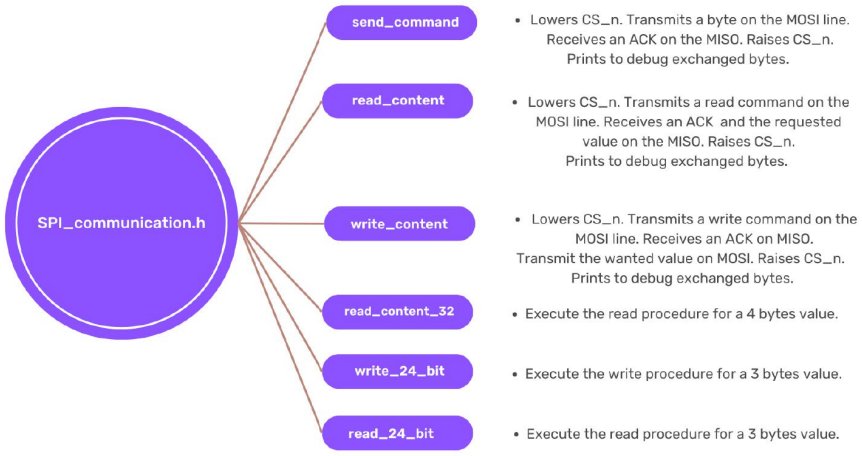


Figure 5.6: SPI_communication file functions.

5.3.1 ADC configuration

Correct communication with the ADC is essential, but it is necessary to know how to format the sequences of bits sent via SPI and which meaning the sequences of 1s and 0s sent have. This information can be found on the device's datasheet and

CMD[7]	CMD[6]	CMD[5]	CMD[4]	CMD[3]	CMD[2]	CMD[1]	CMD[0]
Device Address Bits		Register Address/Fast Command Bits				Command Type Bits	

```
#define MCP_ADDR_M (uint8_t)64
#define MCP_REG_CONFIG0 (uint8_t)(0x1 << 2)
#define MCP_TYPE_IW (uint8_t)2
uint8_t cmd=0;
cmd = (cmd | MCP_ADDR_M | MCP_REG_CONFIG0 | MCP_TYPE_IW );
```

5.4 Bluetooth LE

The latest communication method and peripheral implemented in the firmware is the Bluetooth low energy.
Bluetooth Low Energy (BLE) has emerged as a key wireless communication protocol for low-power, short-range applications. It is particularly suited for battery-operated devices, such as wearables, medical devices, and Internet of Things (IoT) devices, where energy efficiency and data transmission with smartphones or other compatible devices are essential, offering a seamless connectivity and data rate up to 2 Mbps.
The protocol operates in the 2.4 GHz ISM band, providing low-power connectivity for devices. It uses frequency-hopping spread spectrum technology to mitigate interference and ensure reliable communication.
The STM32WB architecture separates the BLE profiles and application, running on the CPU1, from the real-time aspects residing in the BLE peripheral.
The BLE peripheral incorporates a CPU2 processor containing the stack handling the link layer up to the Generic Access Profile (GAP) layers. It also incorporates the physical 2.4 GHz radio.
The application CPU1 collects and computes the data to be transferred to the BLE.
BLE devices operate in two primary roles: the central role and the peripheral role.
The protocol supports short-range communication, typically up to 10 meters, depending on the environment.

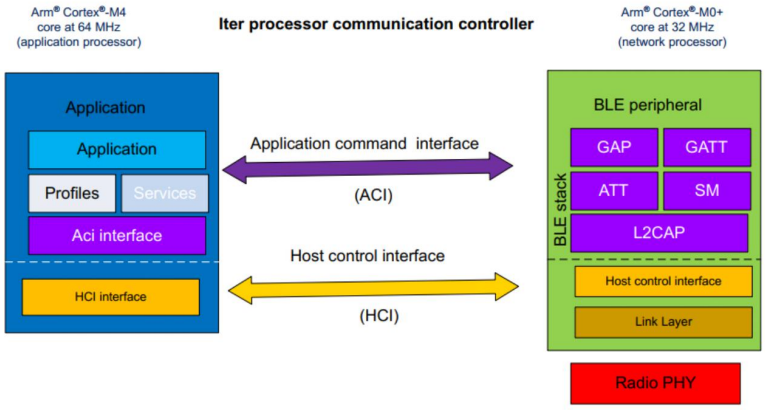


Figure 5.9: Generic Attribute Profile.

Generic Attribute Profile (GATT):

The Generic Attribute Profile (GATT) in Bluetooth Low Energy (BLE) manages the communication between a client and a server device. GATT defines a hierarchical structure consisting of services, characteristics, and descriptors to organize and exchange data.
The GATT client initiates requests to read or write data or subscribe to notifications/indications, and the GATT server responds accordingly.

Services and Characteristics:

Services: Services in GATT represent a collection of related data and behaviors. They define a specific functionality or feature of a device. Services can be predefined by the Bluetooth or custom-defined by developers. Examples of predefined services include the Heart Rate Service, Battery Service, and Device Information Service. Each service has a unique 128-bit Universally Unique Identifier (UUID) that identifies it.

Characteristics: Characteristics are the fundamental data elements within a service. They represent a specific piece of data or a control point for the device. Characteristics have properties, such as read, write, notify, and indicate, which define how they can be accessed or modified. They also have a UUID to identify them uniquely within a service.

Attribute Protocol (ATT): GATT uses the Attribute Protocol (ATT) to manage the exchange of data between the client and server devices. ATT defines the format

and rules for accessing and manipulating characteristics and other attributes. It uses Attribute Handles and Attribute Types (UUIDs) to identify and differentiate attributes.

Descriptors: Descriptors provide additional information or configuration options for characteristics. They provide metadata about characteristics, such as user-friendly descriptions, measurement units, or client configuration settings. Descriptors are optional and can be associated with a characteristic to enhance its functionality or provide contextual information.

In this case, the device was set up with a Server profile offering a customised experience set as follows:

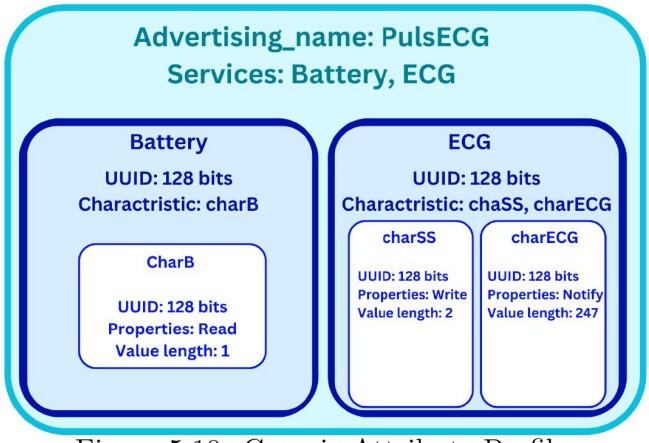


Figure 5.10: Generic Attribute Profile.

As can be seen from the picture above there are two services: Battery and ECG. The Battery service offers only one feature, charB, which is responsible for receiving requests to read the battery status.
The ECG service, on the other hand, offers two features: charSS and charECG.
- charSS takes care of receiving a write request with the time frame for which the ECG signal is to be acquired.
- charECG on notifies the acquired samples to the central device and is characterised by a much larger payload than the usual 20 bytes transmitted as notification.
The management of requests from the bluetooth central device and the exchange of information is managed by the module **custom_stm** whose main functions are shown below.

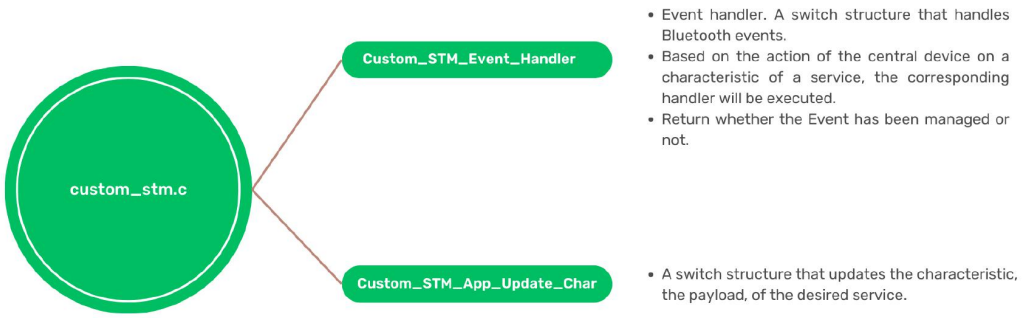


Figure 5.11: custom_stm file functions.

When an event is received, the function *Custom_STM_Event_Handler* is executed. An ID is associated with the event, which maps the service and characteristic to which the event belongs.
The ID is then compared with a switch against the possible verifiable events and, depending on which characteristic generated it, certain functions are executed.
In the firmware, upon receiving a write to the charSS characteristic the function *start_stop_acquisition* of the **PulsECG_core** module will be executed to start or stop the acquisition of ECG signals.
In addition, an opcode will be set to generate longer payloads, up to 247bytes. This is done via the following lines of code:

```
uint32_t ACI_GATT_NOTIFICATION_EXT_EVENT = 0x00400000;
aci_gatt_set_event_mask(ACI_GATT_NOTIFICATION_EXT_EVENT);
aci_gatt_exchange_config(attribute_modified->Connection_Handle);
hci_le_set_data_length(attribute_modified->Connection_Handle, 200, (200+14)*8);
                                                    //TxTime = (payload + 14)*8
```

This reduces the overhead transmitted with each packet and provides 10 times more datarate to be able to send the sampled data in realtime without the need to store it on the device.
In addition, the central device must be alerted to the new payload size with the function:

```
hci_le_write_suggested_default_data_length(200, (200+14)*8); //TxTime = (payload + 14)*8
```

On the other hand, if a read request is received on the charB characteristic, the *send_BLevel* handler of the **PulsECG_core** module will be triggered, which calls the *Custom_STM_App_Update_Char* function and passes it the updated battery value.
This function takes care of updating the characteristic related to a certain service; in fact, it is the same function that updates the characteristic charECG of the service ECG.
In the following, it is shown how are updated the characteristics related to charB (20 bytes) and charECG (200 bytes) in the function *Custom_STM_App_Update_Char*:

```
tBleStatus Custom_STM_App_Update_Char(Custom_STM_Char_Opcode_t CharOpcode, uint8_t *pPayload)
{
  tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
  int ret2=0;

  switch (CharOpcode)
  {
    case CUSTOM_STM_CHARB:
      ret = aci_gatt_update_char_value(CustomContext.CustomBatteryHdle,
                                       CustomContext.CustomCharbHdle,
                                       0, /* charValOffset */
                                       SizeCharb, /* charValueLen */
                                       (uint8_t *)  pPayload);

     case CUSTOM_STM_CHARECG:
      /* USER CODE BEGIN CUSTOM_STM_App_Update_Service_2_Char_2*/
      ret = aci_gatt_update_char_value_ext( 0x0000, CustomContext.CustomEcgHdle,
                  CustomContext.CustomCharecgHdle,
                  0x00, /* DO NOT NOTIFY payload received*/
                  200, /* charValueTotLen */
                    0, /* charValOffset */
                   20, /* charValueLen */
                  (uint8_t *)  pPayload);

      ret2= aci_gatt_update_char_value_ext( 0x0000, CustomContext.CustomEcgHdle,
                  CustomContext.CustomCharecgHdle,
                  0x01, /* NOTIFY payload received*/
                  200, /* charValueTotLen */
                   20, /* charValOffset */
                  180, /* charValueLen */
                  (uint8_t *)(pPayload + 20));

    default:
      break;
  }
  return ret;
}
```

Chapter 6

Digital signal processing

In the realm of Electrocardiography (ECG), Digital Signal Processing (DSP) techniques have proven essential in extracting meaningful information from raw ECG data, aiding in the detection and characterization of cardiac abnormalities.
This chapter focuses on the application of DSP techniques for ECG signal reconstruction, starting from data retrieval via Bluetooth Low Energy (BLE) from the peripheral device, using Python scripts, and culminating in digital filtering algorithms with a special emphasis on their implementation and effectiveness.
By leveraging on the versatility of the "bluetoothctl" Bash command, it becomes possible to establish a connection with the peripheral, retrieve the ECG data, and store it for further processing. This initial step of data retrieval sets the stage for subsequent DSP operations, enabling the reconstruction of the six different ECG signals from the two acquired and provided by the peripheral.
The reconstruction process involves DSP algorithms that employ mathematical techniques to recreate signals, eliminate noise and highlight relevant features for accurate interpretation. In addition, special attention will be paid to the aspect of digital filtering, which is crucial for noise reduction and signal enhancement.
Digital filtering techniques serve as key tools for enhancing the accuracy and reliability of ECG signal reconstruction. By selectively attenuating or eliminating unwanted noise components while preserving the essential features of the ECG waveform, digital filters contribute significantly to the improvement of diagnostic capabilities. The chapter will explore various types of digital filters, including low-pass, high-pass, notch filters, FIR and convolution, discussing their principles, and implementation in Python. The effectiveness of these filters in reducing noise artifacts and enhancing the fidelity of the reconstructed ECG signals will be demonstrated.
From the retrieval of ECG data via BLE to the digital signal filtering techniques, this chapter aims to provide readers with a comprehensive understanding of the DSP pipeline for ECG signal reconstruction.

```
def LPF( data, cutoff, order):

    def butter_lowpass_filter(data, cutoff, fs, order=4):

        b, a = butter(order, cutoff, fs=fs, btype='low', analog=False)
        y = signal.filtfilt(b, a, data)
        return y

    fs = samp_freq       # sample rate, Hz

    y = butter_lowpass_filter(data, cutoff, fs, order)
    return y
```

In the first line, the butter function is called to design a low-pass Butterworth digital filter.The Butterworth filter is characterized by a maximally flat magnitude response in the passband and a gradual roll-off in the stopband.
In the second line, the filtfilt function is used to apply the designed filter to the input signal.

6.2.3 High pass filter

Similarly, a high-pass filter is also applied to accentuate the work done by the analogue front end. The cut-off frequency is 0.5Hz, the order is 3 and it is always a Butterworth filter.

```
def HPF(data, cutoff, order):

    fs = samp_freq       # sample rate, Hz

    b, a = butter(order, cutoff, fs=fs, btype='highpass', analog=False)
    y = signal.filtfilt(b, a, data)

    return y
```

In the first line, the butter function is called to design a high-pass Butterworth digital filter.
In the second line, the filtfilt function is used to apply the designed filter to the input signal.

```
if window_length == None:
    window_length = sample_rate // 10

if window_length % 2 == 0 or window_length == 0: window_length += 1
```

The code first checks if the window length is specified, then if the length is squares regression, and then the smoothed output signal is returned.
Its contribution is very influential so it will not be implemented definitively but rather can be thought of as a beautification filter.

6.3 Final result

Finally, below is an acquisition of the six different ECG signals of a patient with the device studied so far.
The plots show the signals filtered through the analogue front end, sampled with the 24-bit ADC and transmitted in real time to the central device via BLE to then be processed in order to obtain all six leads and improve their readability through additional digital filtering.
The signals acquired with the PulsECG device are compared with those obtained with a professional electrocardiograph, it should be noted that there is a time shift of a few seconds between the two signals.

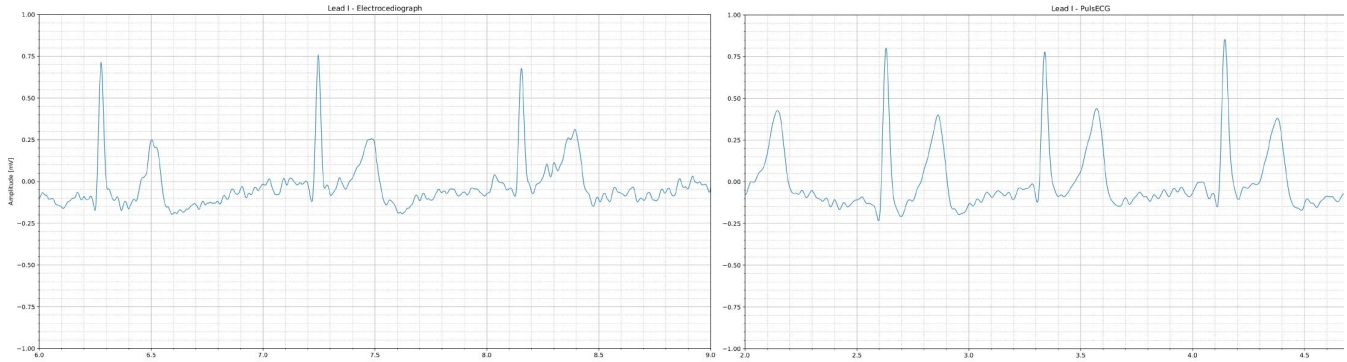


Figure 6.1: Lead I comparison.

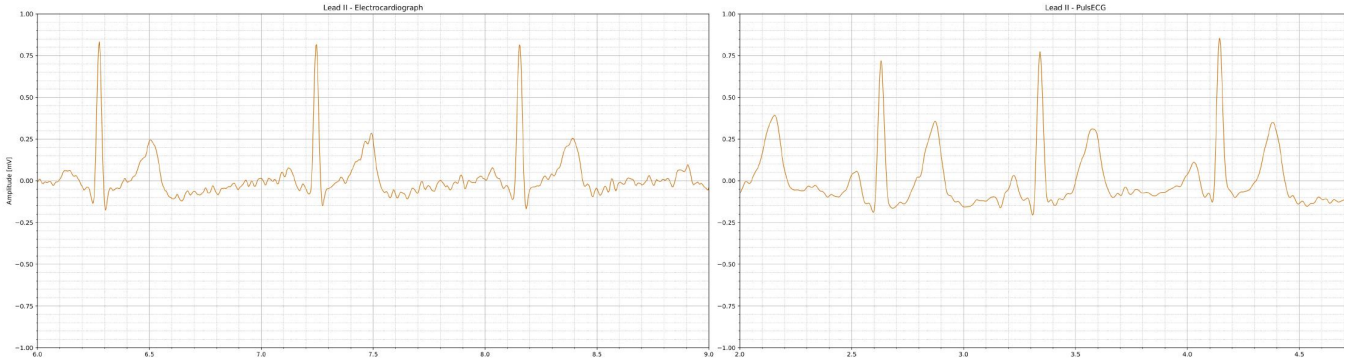


Figure 6.2: Lead II comparison.

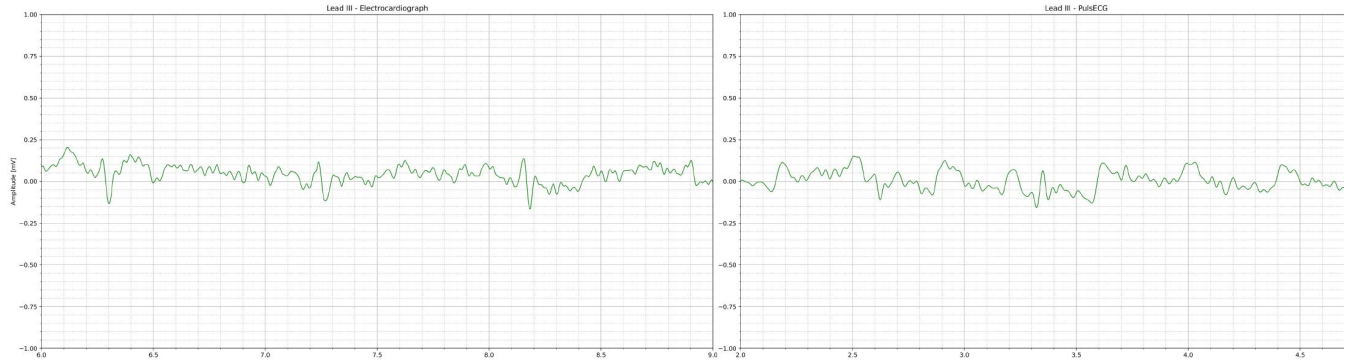


Figure 6.3: Lead III comparison.

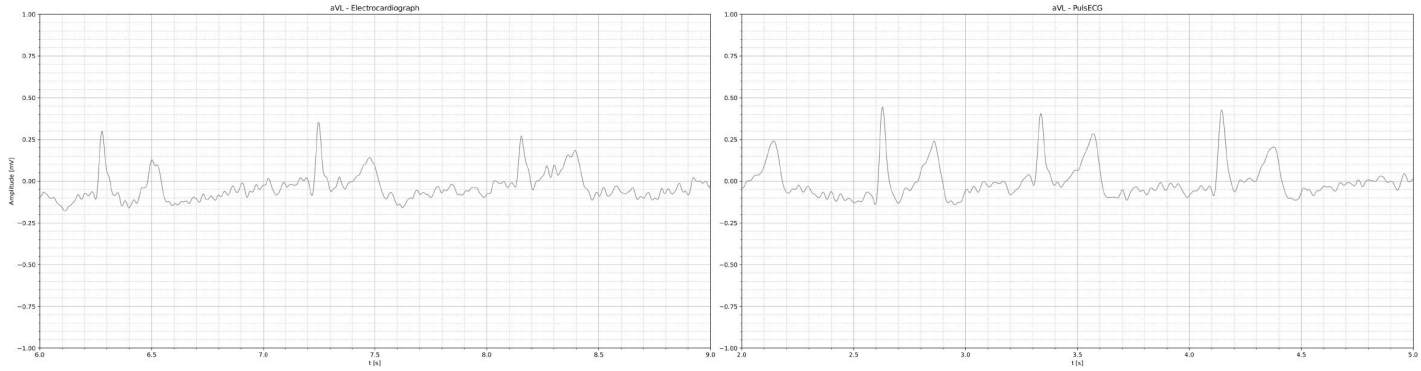


Figure 6.4: Lead aVL comparison.

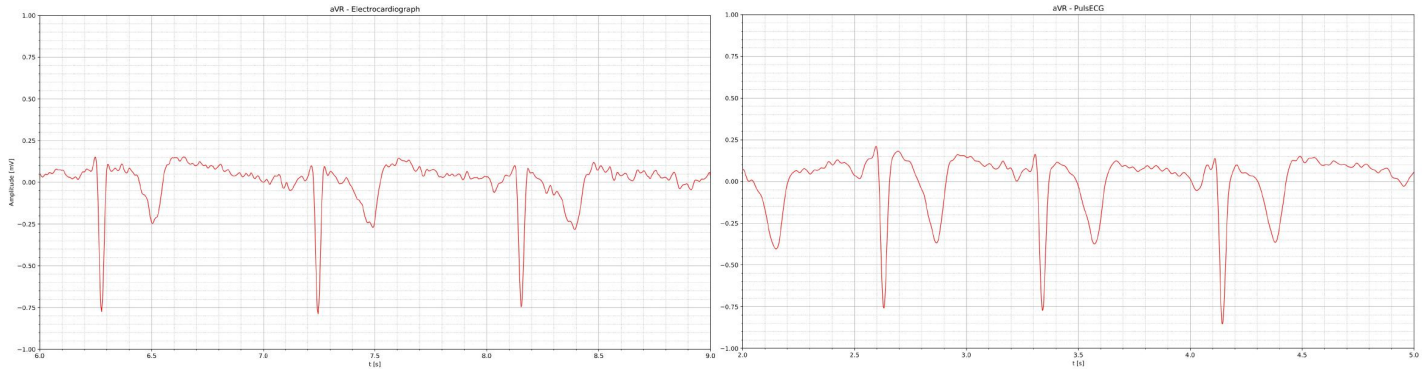


Figure 6.5: Lead aVR comparison.

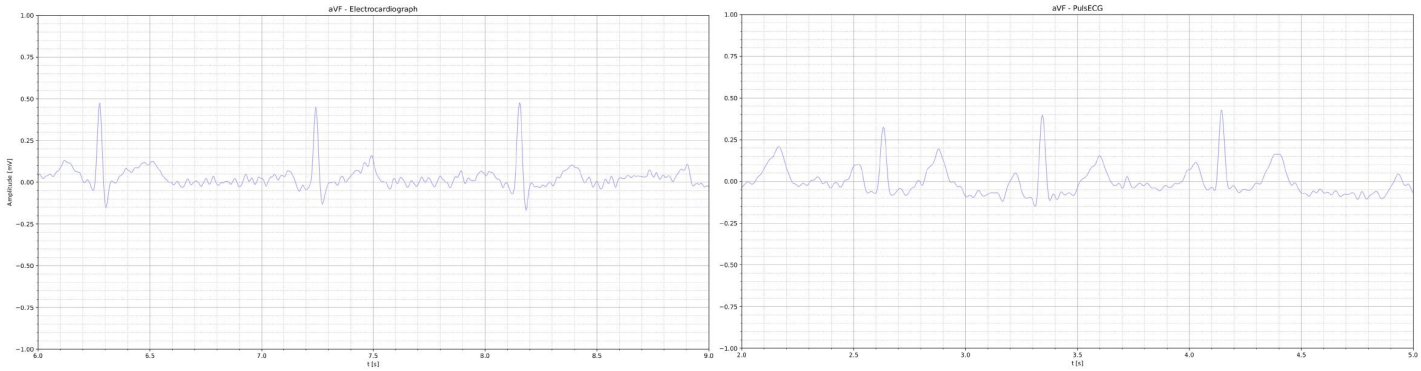


Figure 6.6: Lead aVF comparison.

Appendix A

Firmware code

A.1 main.c

main.c mercoledì 31 maggio 2023, 14:56

```
1 /* USER CODE BEGIN Header */
2 /* USER CODE END Header */
3 /* Includes ------------------------------------------------------------------*/
4 #include "main.h"
5
6 /* Private includes ----------------------------------------------------------*/
7 /* USER CODE BEGIN Includes */
8 #include "custom_app.h"
9 #include "app_conf.h"
10 #include "PulsECG_core.h"
11
12 ADC_HandleTypeDef hadc1;
13
14 IPCC_HandleTypeDef hipcc;
15
16 RTC_HandleTypeDef hrtc;
17
18 SPI_HandleTypeDef hspi1;
19
20 UART_HandleTypeDef huart1;
21
22 /* USER CODE BEGIN PV */
23
24 /* USER CODE END PV */
25
26 /* Private function prototypes -----------------------------------------------*/
27 void SystemClock_Config(void);
28 void PeriphCommonClock_Config(void);
29 static void MX_GPIO_Init(void);
30 static void MX_IPCC_Init(void);
31 static void MX_RF_Init(void);
32 static void MX_RTC_Init(void);
33 static void MX_ADC1_Init(void);
34 static void MX_SPI1_Init(void);
35 static void MX_USART1_UART_Init(void);
36
37
38 int main(void)
39 {
40
41   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
42   HAL_Init();
43   /* Config code for STM32_WPAN (HSE Tuning must be done before system clock
    configuration) */
44   MX_APPE_Config();
45
46   /* Configure the system clock */
47   SystemClock_Config();
48
49 /* Configure the peripherals common clocks */
50   PeriphCommonClock_Config();
51
52   /* IPCC initialisation */
53    MX_IPCC_Init();
54
55   /* Initialize all configured peripherals */
56   MX_GPIO_Init();
57   MX_RF_Init();
58   MX_RTC_Init();
59   MX_ADC1_Init();
60   MX_SPI1_Init();
61   MX_USART1_UART_Init();
62   /* USER CODE BEGIN 2 */
63   /* USER CODE END 2 */
```

```
 64
 65   /* Init code for STM32_WPAN */
 66   MX_APPE_Init();
 67
 68   /* Infinite loop */
 69   /* USER CODE BEGIN WHILE */
 70
 71   uint32_t tempo=HAL_GetTick();
 72   uint8_t tmp=1;
 73   char uart_buf[30];
 74   int uart_buf_len;
 75   uart_buf_len =sprintf(uart_buf, "\tHello_v3\t\r\n");
 76   HAL_UART_Transmit(&huart1, (uint8_t *)uart_buf, uart_buf_len,1000);
 77
 78   //All'accensione eseguire lampeggio led di benvenuto es 10 lampeggi veloci per
    notificare accensione
 79   HAL_Delay(1);
 80
 81   init_PulsECG_core();
 82
 83   while (1)
 84   {
 85     /* USER CODE END WHILE */
 86     MX_APPE_Process();
 87
 88     /* USER CODE BEGIN 3 */
 89
 90   }
 91   /* USER CODE END 3 */
 92 }
 93
 94
 95 /**
 96   * @brief ADC1 Initialization Function
 97   * @param None
 98   * @retval None
 99   */
100 static void MX_ADC1_Init(void)
101 {
102
103   ADC_ChannelConfTypeDef sConfig = {0};
104
105   /** Common config
106   */
107   hadc1.Instance = ADC1;
108   hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
109   hadc1.Init.Resolution = ADC_RESOLUTION_8B;
110   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
111   hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
112   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
113   hadc1.Init.LowPowerAutoWait = DISABLE;
114   hadc1.Init.ContinuousConvMode = DISABLE;
115   hadc1.Init.NbrOfConversion = 1;
116   hadc1.Init.DiscontinuousConvMode = DISABLE;
117   hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
118   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
119   hadc1.Init.DMAContinuousRequests = DISABLE;
120   hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
121   hadc1.Init.OversamplingMode = DISABLE;
122   if (HAL_ADC_Init(&hadc1) != HAL_OK)
123   {
124     Error_Handler();
125   }
126
```

```
127   /** Configure Regular Channel
128   */
129   sConfig.Channel = ADC_CHANNEL_7;
130   sConfig.Rank = ADC_REGULAR_RANK_1;
131   sConfig.SamplingTime = ADC_SAMPLETIME_2CYCLES_5;
132   sConfig.SingleDiff = ADC_SINGLE_ENDED;
133   sConfig.OffsetNumber = ADC_OFFSET_NONE;
134   sConfig.Offset = 0;
135   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
136   {
137     Error_Handler();
138   }
139   /* USER CODE BEGIN ADC1_Init 2 */
140
141   /* USER CODE END ADC1_Init 2 */
142
143 }
144
145
146 /**
147   * @brief SPI1 Initialization Function
148   * @param None
149   * @retval None
150   */
151 static void MX_SPI1_Init(void)
152 {
153
154   hspi1.Instance = SPI1;
155   hspi1.Init.Mode = SPI_MODE_MASTER;
156   hspi1.Init.Direction = SPI_DIRECTION_2LINES;
157   hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
158   hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
159   hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
160   hspi1.Init.NSS = SPI_NSS_SOFT;
161   hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_16;
162   hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
163   hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
164   hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
165   hspi1.Init.CRCPolynomial = 7;
166   hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
167   hspi1.Init.NSSPMode = SPI_NSS_PULSE_ENABLE;
168   if (HAL_SPI_Init(&hspi1) != HAL_OK)
169   {
170     Error_Handler();
171   }
172
173 }
174
175 /**
176   * @brief GPIO Initialization Function
177   * @param None
178   * @retval None
179   */
180 static void MX_GPIO_Init(void)
181 {
182   GPIO_InitTypeDef GPIO_InitStruct = {0};
183
184   /* GPIO Ports Clock Enable */
185   __HAL_RCC_GPIOC_CLK_ENABLE();
186   __HAL_RCC_GPIOB_CLK_ENABLE();
187   __HAL_RCC_GPIOA_CLK_ENABLE();
188   __HAL_RCC_GPIOD_CLK_ENABLE();
189
190   /*Configure GPIO pin Output Level */
```

```
191   HAL_GPIO_WritePin(GPIOB, LED_B_Pin|LED_R_Pin|LD2_Pin|LD3_Pin
192                            |LD1_Pin, GPIO_PIN_RESET);
193
194   /*Configure GPIO pin Output Level */
195   HAL_GPIO_WritePin(GPIOA, EN_ANAL_Pin|CS_ADC_n_Pin, GPIO_PIN_RESET);
196
197   /*Configure GPIO pins : LED_B_Pin LED_R_Pin LD2_Pin LD3_Pin
198                            LD1_Pin */
199   GPIO_InitStruct.Pin = LED_B_Pin|LED_R_Pin|LD2_Pin|LD3_Pin
200                            |LD1_Pin;
201   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
202   GPIO_InitStruct.Pull = GPIO_NOPULL;
203   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
204   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
205
206   /*Configure GPIO pins : EN_ANAL_Pin CS_ADC_n_Pin */
207   GPIO_InitStruct.Pin = EN_ANAL_Pin|CS_ADC_n_Pin;
208   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
209   GPIO_InitStruct.Pull = GPIO_NOPULL;
210   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
211   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
212
213   /*Configure GPIO pin : DRY_n_Pin */
214   GPIO_InitStruct.Pin = DRY_n_Pin;
215   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
216   GPIO_InitStruct.Pull = GPIO_PULLUP;
217   HAL_GPIO_Init(DRY_n_GPIO_Port, &GPIO_InitStruct);
218
219   /*Configure GPIO pin : WKUP_BTN_Pin */
220   GPIO_InitStruct.Pin = WKUP_BTN_Pin;
221   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
222   GPIO_InitStruct.Pull = GPIO_NOPULL;
223   HAL_GPIO_Init(WKUP_BTN_GPIO_Port, &GPIO_InitStruct);
224
225   /*Configure GPIO pin : B1_Pin */
226   GPIO_InitStruct.Pin = B1_Pin;
227   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
228   GPIO_InitStruct.Pull = GPIO_PULLUP;
229   HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
230
231   /*Configure GPIO pins : PA11 PA12 */
232   GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_12;
233   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
234   GPIO_InitStruct.Pull = GPIO_NOPULL;
235   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
236   GPIO_InitStruct.Alternate = GPIO_AF10_USB;
237   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
238
239   /*Configure GPIO pins : B2_Pin B3_Pin */
240   GPIO_InitStruct.Pin = B2_Pin|B3_Pin;
241   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
242   GPIO_InitStruct.Pull = GPIO_NOPULL;
243   HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
244
245 }
246
247
```

A.2 PulsECG_core.h

PulsECG_core.h mercoledì 31 maggio 2023, 12:12

```
/*
 * PulsECG_core.h
 *
 *  Created on: Apr 27, 2023
 *      Author: marco
 */

#ifndef APP_PULSECG_CORE_H_
#define APP_PULSECG_CORE_H_

#include <ADC_command.h>
#include <SPI_communication.h>
#include "CallBack.h"
#include "custom_stm.h"
#include "custom_app.h"
#include "stm32_seq.h"


#define ADC_BYTES 1000
#define NOTIFY_PAYLOAD 200

static uint32_t duration, t_start, BTime;
static uint8_t BLevel, overflow;
static uint8_t bool_ECGreading, bool_BATreading, bool_already_read;
static uint8_t ADCDATA[ADC_BYTES];
static uint16_t index_1, index_2, tot_sample;
extern uint16_t tmp_BT;
static uint32_t NotifyECGval;


void init_PulsECG_core(void);
void update_BLevel(void);
void send_BLevel(void);
void start_stop_acquisition(uint8_t tempo);
void arm_board(void);
void disarm_board(void);
void acquire(void);
void sendSample(void);

void BT_config(void);


#endif /* APP_PULSECG_CORE_H_ */

```

A.3 PulsECG_core.c

A.4 ADC_command.h


```
1 /*
2  * ADC.h
3  *
4  *  Created on: Nov 19, 2022
5  *      Author: marco
6  */
7
8 #ifndef INC_ADC_COMMAND_H_
9 #define INC_ADC_COMMAND_H_
10
11 #include <MCP3562.h>
12 #include <SPI_communication.h>
13 #include "CallBack.h"
14
15 void adc_setup(void);
16
17 void adc_reset();
18
19 void adc_set_CONFIG0(int w_val);
20
21 void adc_set_CONFIG3(int w_val);
22
23 void adc_read_CONFIG1();
24
25 void adc_set_CONFIG1(int w_val);
26
27 void adc_set_MUX();
28
29 void adc_set_SCAN();
30
31 void adc_get_value(int mode, int seconds);
32
33 void adc_start_restart();
34
35 void adc_get_calibration_offset();
36
37 #endif /* INC_ADC_COMMAND_H_ */
38
```

A.5 ADC_command.c

ADC_command.c mercoledì 31 maggio 2023, 12:05

```c
/*
 * ADC.c
 *
 *  Created on: Nov 19, 2022
 *      Author: marco
 */

#ifndef SRC_ADC_C_
#define SRC_ADC_C_

#include <ADC_command.h>

static uint8_t cmd=0, write_val=0;


void adc_continous_read(void){

    uint8_t cmd = (cmd | MCP_ADDR_M | MCP_REG_ADCDATA | MCP_TYPE_SR );

    HAL_GPIO_WritePin(CS_ADC_n_GPIO_Port, CS_ADC_n_Pin, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi1, (uint8_t *)&cmd, 1, 100);

}


void adc_setup(void){
    adc_reset();
    //adc_start_restart();

    adc_set_CONFIG0(-1);
    adc_set_CONFIG1(-1);    // OSR = 1024 = 0x14
    //adc_read_CONFIG1();

    adc_set_CONFIG3(0xD0); //0xD2 for continous read
    //adc_set_MUX();
    adc_set_SCAN();
    adc_start_restart();
}


void adc_reset(){
    char cmdName[32]="RESET";

    cmd=0;
    cmd = (cmd | MCP_ADDR_M | MCP_RESET); // COMMAND 01.111000

    send_command(cmdName,32,cmd);
}



void adc_set_CONFIG0(int w_val){

    char cmdName[32]="CONFIG0";
    cmd=0; write_val=0;

    //write the register
    cmd = (cmd | MCP_ADDR_M | MCP_REG_CONFIG0 | MCP_TYPE_IW );

    if (w_val!=-1) write_val= (write_val | (uint8_t) w_val );
    else write_val= (write_val | (uint8_t) 0xC3 );  //default value

    write_content(cmdName,32,cmd, write_val);

```

A.6 SPI_communication.h

SPI_communication.h mercoledì 31 maggio 2023, 12:12

```c
/*
 * communication.h
 *
 *  Created on: Nov 19, 2022
 *      Author: marco
 */

#ifndef INC_SPI_COMMUNICATION_H_
#define INC_SPI_COMMUNICATION_H_


#include <MCP3562.h>
#include "CallBack.h"

void toBinary(int num, char binary[]);

void print_content(int num, int type);

//utility function: send the command and read the ack
void u_send_command(uint8_t *cmd, uint8_t *bin_send);

void send_command(char *cmdName, size_t lengthCmdName, uint8_t cmd);

//Get the content of a register
void read_content(char *cmdName, size_t lengthCmdName, uint8_t cmd);

void read_content_32(char *cmdName, size_t lengthCmdName, uint8_t cmd);

void write_24_bit(char *cmdName, size_t lengthCmdName, uint8_t cmd, uint8_t reg_val
[]);

void read_24_bit(char *cmdName, size_t lengthCmdName, uint8_t cmd);

//Write something to a register
void write_content(char *cmdName, size_t lengthCmdName, uint8_t cmd, uint8_t new_val
);


#endif /* INC_SPI_COMMUNICATION_H_ */

```

A.7 SPI_communication.c


```c
/*
 * communication.c
 *
 *  Created on: Nov 19, 2022
 *      Author: marco
 */


#ifndef INC_COMMUNICATION_C_
#define INC_COMMUNICATION_C_

#include <SPI_communication.h>

static char uart_buf[90];
static int uart_buf_len;


void toBinary(int num, char binary[])
{
    for (int i=0; i<8; i++) {

        binary[7-i] = (char)((num >> i & 1) + '0');
        binary[8]='\0';
    }
}


//type: 1 = ack | 2 = read value | 3 = wrote value
void print_content(int num, int type){
    char binary[34]={'\0'};
    toBinary(num, binary);

    if(type==1)      uart_buf_len = sprintf(uart_buf, "\tACK received:\t %s aka %u \r\n", binary, num);
    else if(type==2) uart_buf_len = sprintf(uart_buf, "\tContent read:\t %s aka %u \r\n", binary, num);
    else if(type==3) uart_buf_len = sprintf(uart_buf, "\tContent wrote:\t %s aka %u \r\n", binary, num);

    HAL_UART_Transmit(&huart1, (uint8_t *)uart_buf, uart_buf_len, 100);
}



//Utility function: send the command and return the ack
//First function called for every communication
void u_send_command(uint8_t *cmd, uint8_t *recev){

    char bin_send[32]={'\0'};
    HAL_Delay(1);

    toBinary(*cmd, bin_send);

    uart_buf_len = sprintf(uart_buf, "\tbyte sent:\t %s aka %u \r\n", bin_send, *cmd);
    HAL_UART_Transmit(&huart1, (uint8_t *)uart_buf, uart_buf_len, 100);

    HAL_SPI_TransmitReceive(&hspi1, (uint8_t *)cmd, (uint8_t *)recev, 1, 100);

}



//For commands that don't require additional read or write
```

A.9 custom_stm.h


```
 1
 2 /* Define to prevent recursive inclusion -------------------------------------*/
 3 #ifndef __CUSTOM_STM_H
 4 #define __CUSTOM_STM_H
 5
 6 #ifdef __cplusplus
 7 extern "C" {
 8 #endif
 9
10 /* Includes ------------------------------------------------------------------*/
11 /* USER CODE BEGIN Includes */
12
13 /* USER CODE END Includes */
14
15 /* Exported types ------------------------------------------------------------*/
16 typedef enum
17 {
18   /* Battery */
19   CUSTOM_STM_CHARB,
20   /* ECG */
21   CUSTOM_STM_CHARSS,
22   CUSTOM_STM_CHARECG,
23 } Custom_STM_Char_Opcode_t;
24
25 typedef enum
26 {
27   /* charB */
28   CUSTOM_STM_CHARB_READ_EVT,
29   /* charSS */
30   CUSTOM_STM_CHARSS_WRITE_EVT,
31   /* charECG */
32   CUSTOM_STM_CHARECG_READ_EVT,
33   CUSTOM_STM_CHARECG_NOTIFY_ENABLED_EVT,
34   CUSTOM_STM_CHARECG_NOTIFY_DISABLED_EVT,
35
36   CUSTOM_STM_BOOT_REQUEST_EVT
37 } Custom_STM_Opcode_evt_t;
38
39 typedef struct
40 {
41   uint8_t * pPayload;
42   uint8_t   Length;
43 } Custom_STM_Data_t;
44
45 typedef struct
46 {
47   Custom_STM_Opcode_evt_t       Custom_Evt_Opcode;
48   Custom_STM_Data_t             DataTransfered;
49   uint16_t                      ConnectionHandle;
50   uint8_t                       ServiceInstance;
51 } Custom_STM_App_Notification_evt_t;
52
53 /* USER CODE BEGIN ET */
54
55 /* USER CODE END ET */
56
57 /* Exported constants --------------------------------------------------------*/
58 extern uint8_t SizeCharb;
59 extern uint8_t SizeCharss;
60 extern uint8_t SizeCharecg;
61
62 /* USER CODE BEGIN EC */
63
64 /* USER CODE END EC */
```

A.10 custom_stm.c

custom_stm.c mercoledì 31 maggio 2023, 15:55

```
 1
 2 /* Includes ------------------------------------------------------------------*/
 3 #include "common_blesvc.h"
 4 #include "custom_stm.h"
 5
 6 /* USER CODE BEGIN Includes */
 7 #include "main.h"
 8 #include "PulsECG_core.h"
 9 /* USER CODE END Includes */
10
11 /* Private typedef -----------------------------------------------------------*/
12 typedef struct{
13   uint16_t  CustomBatteryHdle;                    /**< Battery handle */
14   uint16_t  CustomCharbHdle;                    /**< charB handle */
15   uint16_t  CustomEcgHdle;                    /**< ECG handle */
16   uint16_t  CustomCharssHdle;                    /**< charSS handle */
17   uint16_t  CustomCharecgHdle;                    /**< charECG handle */
18 }CustomContext_t;
19
20
21 /* Private macros ------------------------------------------------------------*/
22 #define CHARACTERISTIC_DESCRIPTOR_ATTRIBUTE_OFFSET         2
23 #define CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET              1
24 /* USER CODE BEGIN PM */
25
26 /* USER CODE END PM */
27
28 /* Private variables ---------------------------------------------------------*/
29 uint8_t SizeCharb = 1;
30 uint8_t SizeCharss = 2;
31 uint8_t SizeCharecg = 247;
32
33 /**
34  * START of Section BLE_DRIVER_CONTEXT
35  */
36 PLACE_IN_SECTION("BLE_DRIVER_CONTEXT") static CustomContext_t CustomContext;
37
38 /**
39  * END of Section BLE_DRIVER_CONTEXT
40  */
41
42 /* USER CODE BEGIN PV */
43 uint16_t tmp_BT=0;
44 /* USER CODE END PV */
45
46 /* Private function prototypes -----------------------------------------------*/
47 static SVCCTL_EvtAckStatus_t Custom_STM_Event_Handler(void *pckt);
48
49 static SVCCTL_EvtAckStatus_t Custom_STM_Event_Handler(void *Event)
50 {
51   SVCCTL_EvtAckStatus_t return_value;
52   hci_event_pckt *event_pckt;
53   evt_blecore_aci *blecore_evt;
54   aci_gatt_attribute_modified_event_rp0 *attribute_modified;
55   aci_gatt_read_permit_req_event_rp0    *read_req;
56   Custom_STM_App_Notification_evt_t     Notification;
57   /* USER CODE BEGIN Custom_STM_Event_Handler_1 */
58
59   /* USER CODE END Custom_STM_Event_Handler_1 */
60
61   return_value = SVCCTL_EvtNotAck;
62   event_pckt = (hci_event_pckt *)(((hci_uart_pckt*)Event)->data);
63
64   switch (event_pckt->evt)
```

```
65  {
66    case HCI_VENDOR_SPECIFIC_DEBUG_EVT_CODE:
67      blecore_evt = (evt_blecore_aci*)event_pckt->data;
68      switch (blecore_evt->ecode)
69      {
70        case ACI_GATT_ATTRIBUTE_MODIFIED_VSEVT_CODE:
71          /* USER CODE BEGIN EVT_BLUE_GATT_ATTRIBUTE_MODIFIED_BEGIN */
72
73          /* USER CODE END EVT_BLUE_GATT_ATTRIBUTE_MODIFIED_BEGIN */
74          attribute_modified = (aci_gatt_attribute_modified_event_rp0*)blecore_evt-
   >data;
75          if (attribute_modified->Attr_Handle == (CustomContext.CustomCharecgHdle +
   CHARACTERISTIC_DESCRIPTOR_ATTRIBUTE_OFFSET))
76          {
77            return_value = SVCCTL_EvtAckFlowEnable;
78            /* USER CODE BEGIN CUSTOM_STM_Service_2_Char_2 */
79
80            /* USER CODE END CUSTOM_STM_Service_2_Char_2 */
81            switch (attribute_modified->Attr_Data[0])
82            {
83
84              case (!(COMSVC_Notification)):
85                /* USER CODE BEGIN CUSTOM_STM_Service_2_Char_2_Disabled_BEGIN */
86
87                /* USER CODE END CUSTOM_STM_Service_2_Char_2_Disabled_BEGIN */
88                Notification.Custom_Evt_Opcode =
   CUSTOM_STM_CHARECG_NOTIFY_DISABLED_EVT;
89                Custom_STM_App_Notification(&Notification);
90                /* USER CODE BEGIN CUSTOM_STM_Service_2_Char_2_Disabled_END */
91
92                /* USER CODE END CUSTOM_STM_Service_2_Char_2_Disabled_END */
93                break;
94
95              /* Enabled Notification management */
96              case COMSVC_Notification:
97                /* USER CODE BEGIN
   CUSTOM_STM_Service_2_Char_2_COMSVC_Notification_BEGIN */
98
99                /* USER CODE END
   CUSTOM_STM_Service_2_Char_2_COMSVC_Notification_BEGIN */
100               Notification.Custom_Evt_Opcode =
   CUSTOM_STM_CHARECG_NOTIFY_ENABLED_EVT;
101               Custom_STM_App_Notification(&Notification);
102               break;
103
104             default:
105             break;
106           }
107         }  /* if (attribute_modified->Attr_Handle ==
   (CustomContext.CustomCharecgHdle + CHARACTERISTIC_DESCRIPTOR_ATTRIBUTE_OFFSET))*/
108
109         else if (attribute_modified->Attr_Handle ==
   (CustomContext.CustomCharssHdle + CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))
110         {
111           return_value = SVCCTL_EvtAckFlowEnable;
112           /* USER CODE BEGIN
   CUSTOM_STM_Service_2_Char_1_ACI_GATT_ATTRIBUTE_MODIFIED_VSEVT_CODE */
113
114           uint8_t *ptr = attribute_modified->Attr_Data;
115           uint8_t duration=0;
116
117           duration = (ptr[0]-'0')*10 + (ptr[1]-'0');
118
119           uint32_t ACI_GATT_NOTIFICATION_EXT_EVENT = 0x00400000;
```

```
120             aci_gatt_set_event_mask(ACI_GATT_NOTIFICATION_EXT_EVENT);
121             aci_gatt_exchange_config(attribute_modified->Connection_Handle);
122             hci_le_set_data_length(attribute_modified->Connection_Handle, 200,
    (200+14)*8);
123             hci_le_write_suggested_default_data_length(200, (200+14)*8);
124 
125             start_stop_acquisition(duration);
126 
127           } /* if (attribute_modified->Attr_Handle ==
    (CustomContext.CustomCharssHdle + CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))*/
128 
129           break;
130 
131         case ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE :
132           /* USER CODE BEGIN EVT_BLUE_GATT_READ_PERMIT_REQ_BEGIN */
133 
134           /* USER CODE END EVT_BLUE_GATT_READ_PERMIT_REQ_BEGIN */
135           read_req = (aci_gatt_read_permit_req_event_rp0*)blecore_evt->data;
136           if (read_req->Attribute_Handle == (CustomContext.CustomCharbHdle +
    CHARACTERISTIC_VALUE_ATTRIBUTE_OFFSET))
137           {
138             return_value = SVCCTL_EvtAckFlowEnable;
139             /*USER CODE BEGIN
    CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_1 */
140 
141              send_BLevel();
142 
143             /*USER CODE END
    CUSTOM_STM_Service_1_Char_1_ACI_GATT_READ_PERMIT_REQ_VSEVT_CODE_1*/
144             aci_gatt_allow_read(read_req->Connection_Handle);
145           }
146           break;
147 
148         case ACI_GATT_WRITE_PERMIT_REQ_VSEVT_CODE:
149           break;
150         default:
151           break;
152       }
153       break; /* HCI_VENDOR_SPECIFIC_DEBUG_EVT_CODE */
154 
155 
156     default:
157       break;
158   }
159 
160   return(return_value);
161 }/* end Custom_STM_Event_Handler */
162 
163 /**
164  * @brief  Characteristic update
165  * @param  CharOpcode: Characteristic identifier
166  * @param  Service_Instance: Instance of the service to which the characteristic
    belongs
167  *
168  */
169 tBleStatus Custom_STM_App_Update_Char(Custom_STM_Char_Opcode_t CharOpcode, uint8_t
    *pPayload)
170 {
171   tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
172   /* USER CODE BEGIN Custom_STM_App_Update_Char_1 */
173 
174   int ret2=0;
175 
176   /* USER CODE END Custom_STM_App_Update_Char_1 */
```

```
177
178   switch (CharOpcode)
179   {
180
181     case CUSTOM_STM_CHARB:
182       ret = aci_gatt_update_char_value(CustomContext.CustomBatteryHdle,
183                                        CustomContext.CustomCharbHdle,
184                                        0, /* charValOffset */
185                                        SizeCharb, /* charValueLen */
186                                        (uint8_t *)  pPayload);
187       if (ret != BLE_STATUS_SUCCESS)
188       {
189         APP_DBG_MSG("  Fail   : aci_gatt_update_char_value CHARB command, result :
    0x%x \n\r", ret);
190       }
191       else
192       {
193         APP_DBG_MSG("  Success: aci_gatt_update_char_value CHARB command\n\r");
194       }
195       /* USER CODE BEGIN CUSTOM_STM_App_Update_Service_1_Char_1*/
196
197       /* USER CODE END CUSTOM_STM_App_Update_Service_1_Char_1*/
198       break;
199
200     case CUSTOM_STM_CHARSS:
201       ret = aci_gatt_update_char_value(CustomContext.CustomEcgHdle,
202                                        CustomContext.CustomCharssHdle,
203                                        0, /* charValOffset */
204                                        SizeCharss, /* charValueLen */
205                                        (uint8_t *)  pPayload);
206       if (ret != BLE_STATUS_SUCCESS)
207       {
208         APP_DBG_MSG("  Fail   : aci_gatt_update_char_value CHARSS command, result :
    0x%x \n\r", ret);
209       }
210       else
211       {
212         APP_DBG_MSG("  Success: aci_gatt_update_char_value CHARSS command\n\r");
213       }
214
215       break;
216
217     case CUSTOM_STM_CHARECG:
218
219       ret = aci_gatt_update_char_value_ext( 0x0000, CustomContext.CustomEcgHdle,
    CustomContext.CustomCharecgHdle,
220                   0x00,
221                   200,
222                    0,
223                   20,
224                   (uint8_t *)  pPayload);
225       ret2= aci_gatt_update_char_value_ext( 0x0000, CustomContext.CustomEcgHdle,
    CustomContext.CustomCharecgHdle,
226                   0x01,
227                   200,
228                   20,
229                   180,
230                   (uint8_t *)(pPayload + 20));
231
232             if (ret != BLE_STATUS_SUCCESS || ret2 != BLE_STATUS_SUCCESS)
233             {
234               APP_DBG_MSG("  Fail   : aci_gatt_update_char_value CHARECG command,
    result : 0x%x \n\r", ret);
235             }
```

```
236             else
237             {
238               APP_DBG_MSG("  Success: aci_gatt_update_char_value CHARECG command\n
    \r");
239             }
240       if (ret == BLE_STATUS_SUCCESS && ret2 == BLE_STATUS_SUCCESS) tmp_BT++;
241 
242       break;
243 
244     default:
245       break;
246   }
247 
248   return ret;
249 }
250 
```