*Prof. Martino Giaquinto*

# Project Work Embedded Systems for E-Health:

Apicella Mario Bruno Salvatore Fiorillo Emanuela Schettini Domenico

June 11, 2024


UNIVERSITÀ
DEGLI STUDI
DI SALERNO

# Contents

# 1 Introduction

In the modern medical context, the use of embedded systems for monitoring of patients has become crucial. Technological advancement has led to the development of sophisticated devices that can achieve and analyze vital data in real time thereby easing the burden of healthcare professionals and improving the quality of care given to patients. It poses a big logistic challenge to be able to monitor one's vital parameters continuously. It is impractical, if not impossible, for a doctor to be always vigilant about the physiological parameters of his patients for the whole day. For this reason, the use of sophisticated medical devices, capable of autonomously recording and transmitting data, is a milestone within the health sector.

These devices not only enhance clinical monitoring but also allow timely intervention in case of anomalies, potentially saving lives. Among the continuous monitoring devices, the Holter monitor is highly valuable but requires medical personnel for setup and interpretation. Meanwhile, wearable devices like smartwatches have emerged, offering affordable health monitoring. These devices include sensors for heart rate, blood oxygen levels, sleep quality, and activity tracking. They can connect to smartphones for easy data access and sharing with doctors. Additionally, a range of remote health monitoring technologies exists, from wearable sensors to implantable devices, with accuracy varying by design.

This project aims to explore in detail the design and development of an embedded device for continuous monitoring of vital signs. Both the hardware aspects, including the necessary sensors and electronic components, and the software aspects, which include data processing, user interface and secure transmission of information, will be addressed. Through an analysis of existing technologies and the implementation of innovative solutions, the goal is to create a device that is not only effective and reliable, but also accessible and easy to use for patients.

**Our group hopes that this report conveys the hard work and passion we dedicated to producing the best possible outcome for this project. Enjoy!!**

2

# Solution

During our **Embedded System For-E health** course, the Professor talked extensively about the critical role of medical devices in modern healthcare. He emphasized how embedded systems are revolutionizing the field of digital medicine by providing real-time monitoring, diagnostic capabilities, and even therapeutic interventions. One of the key points discussed was the integration of sensors and microcontrollers in medical devices. These components are essential for collecting physiological data such as heart rate, blood pressure, glucose levels, and other vital signs. The Professor highlighted how these devices not only gather data but also process and analyze it on the spot, enabling immediate feedback and decision-making.

## 2.1 The Project: requests

The proposal made by the Professor to us is to **realize a wearable system for health monitoring mainly based on:**

- **Acquisition and elaboration of ECG signals**: This involves capturing the electrical activity of the heart to monitor cardiac health, detect abnormalities, and assist in diagnosing conditions such as arrhythmias.
- **Acquisition and elaboration of PPG signals**: This entails using photoplethysmography to measure blood volume changes in the microvascular bed of tissue, providing valuable information on heart rate.

### 2.1.1 Main Specifications

We were required to implement three different modalities in which our system can work:

- **Home Mode:** Visualize Heart Rate (HR) and HR variability (HRV) on an OLED display.
- **Waveform monitoring:** Monitoring on OLED display of ECG and PPG signals
- **Advanced:** Features trend monitoring.

In order to skip from a mode to another we were asked to use the push button.

The **Home mode**, allows visualizing Heart Rate (HR) and Heart Rate Variability (HRV) on an OLED display. In this mode, it is possible to choose to use only one of the sensors, justifying the choice. The values should be updated every 30 seconds, and only reliable values should be displayed. Additionally, the system should show messages indicating incorrect contact with the sensor. For each measurement, a message should be sent via UART containing the date/time and detailed information about the measurements taken, such as active sensors, duration of the time window, and the number of samples acquired. The display and UART should also show alert messages indicating excessively high HR and/or HRV values. HR can be calculated as the inverse of the time interval between two R peaks of the ECG or between two systolic peaks of the PPG. HRV can be calculated as the standard deviation of the time intervals between successive R peaks or systolic peaks. If high beats are detected, an alert represented by a heart symbol is displayed on the screen.


Figure 1: Home mode

The second operating mode is **Waveform Monitoring**, which involves monitoring ECG and PPG signals on an OLED display. The displayed waveforms can optionally be filtered to eliminate noise. The system should send the raw samples acquired in real-time via UART, formatting the data into two columns, one for each signal (ECG and PPG). Additionally, an LED should be turned on with an intensity that follows the heart rate.
The third operating mode is **Advanced**, which includes features for trend monitoring.

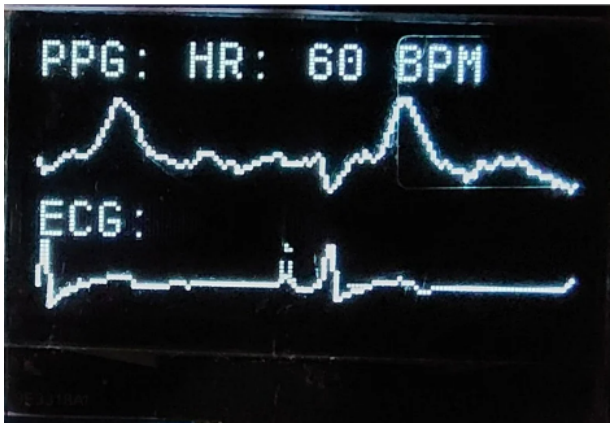



Figure 2: Waveform mode

In this mode, the system should graphically represent the temporal trends of Heart Rate (HR) and Heart Rate Variability (HRV) with updates every 5 seconds. Additionally, the system should attempt to extract other relevant features such as the systolic/diastolic time duration (from the PPG signal) and the Pulse Arrival Time (PAT), which is the temporal delay between an R peak of the ECG and the corresponding systolic peak of the PPG signal.


Figure 3: Advanced mode

## 2.2 Work Context

Considering the requirements assigned to us, we began planning our project. The first thing we considered was how to implement all the sensors in an original device that could potentially be used in real life.

We spent a lot of time thinking about how to create something tangible and usable by everyone. The solution was to create something similar to a smartwatch (we'll explain how in the following chapters).

Obviously the device must be very accurate and easy to use by everyone. To achieve this, we focused on user-friendly design principles and precision engineering. Our aim is to ensure that the device not only meets the technical requirements but also provides a seamless and intuitive user experience. In the following chapters, we will go deeper into the specific features and functionalities that make this possible.

## 2.3 The Device

The device we're gonna create is an Embedded System made obviously by **a circuit and a firmware** designed by us. The main board is the **NUCLEO-G474RE** provided by **STM**.

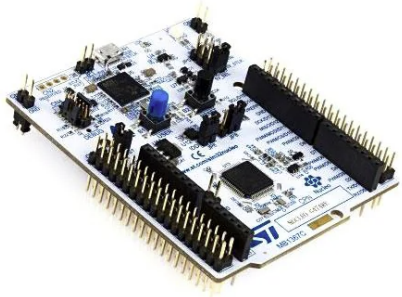


Figure 4: STM: NUCLEO-G474RE

In conjunction with the board, we will utilize several sensors that will be essential for acquiring the necessary information. Specifically, we will use:

- **AD8232 Heart Monitor Module**: This module will help in monitoring the heart's electrical activity, providing crucial ECG data.
- **PulseSensor**: This sensor will measure the user's heart rate by detecting blood volume changes.
- **Real-Time Clock DS3231**: This module will keep track of time, ensuring that our data is timestamped accurately.
- **OLED Display SSD1306**: A small, efficient display that will present real-time data to the user in a clear and readable format.
- **Development KIT (Breadboard, resistors, LEDs, push buttons)**: A collection of essential components for prototyping and testing our circuit designs.

# 3 Project Design

Let us begin with the completion of the final device, starting with our design specifications. These specifications include both hardware and software sections.

Regarding the hardware aspect, we meticulously selected only the essential components to ensure smooth and efficient operation. This involved tasks such as positioning the sensors accurately, ensuring reliable connections between them, and developing a compact design that could be easily understood by anyone. The goal was to create a device that is both functional and user-friendly.

On the software side, we focused on developing algorithms that would facilitate efficient interaction between the inputs and outputs of the sensors. This included ensuring real-time data processing and seamless communication between the hardware components and the user interface.

Additionally, we integrated features that enhance the overall user experience, such as intuitive controls and clear data visualization. Our efforts were aimed at achieving a balance between performance and usability, ensuring that the device meets the needs of both end-users and healthcare professionals.

## 3.1 State Flow Diagram

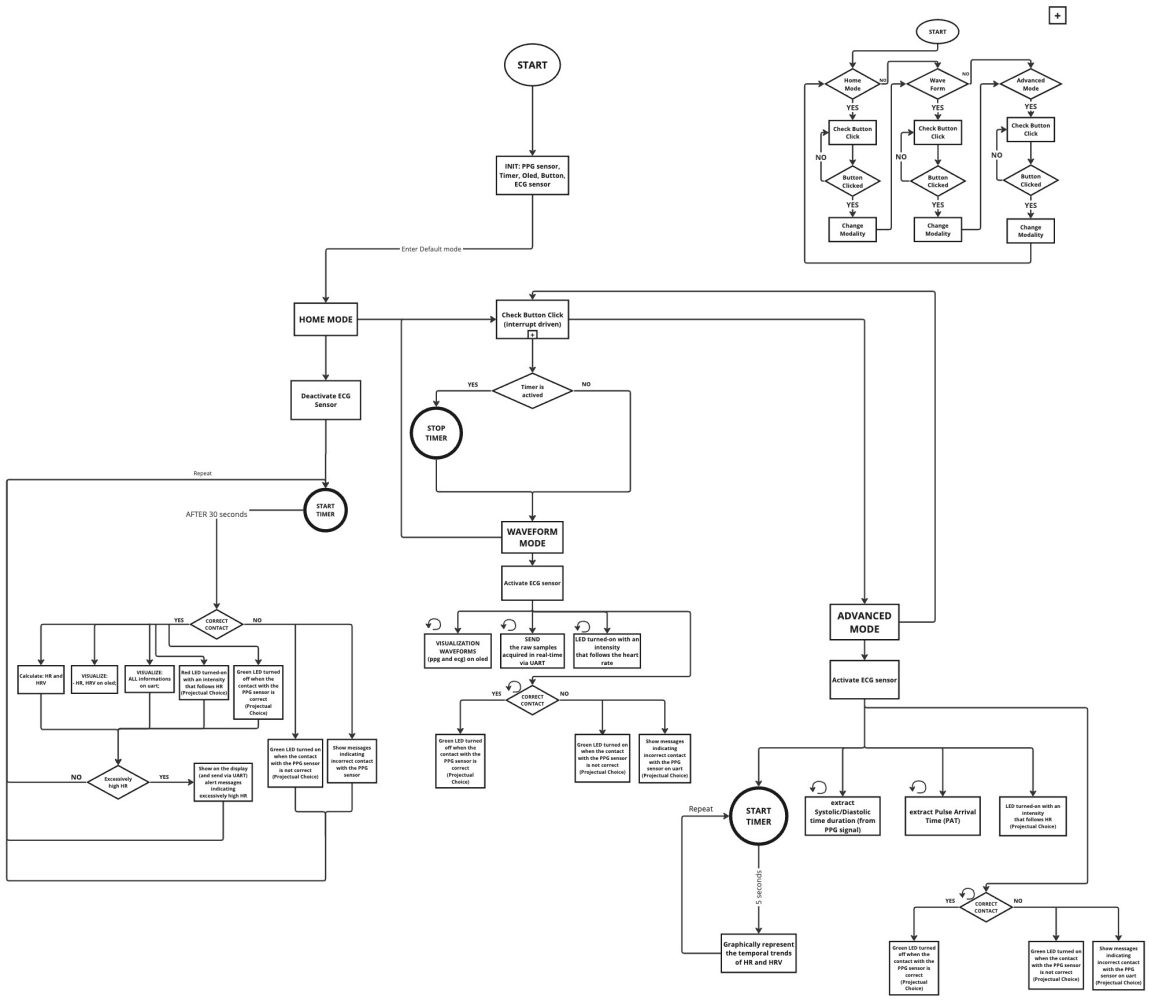



Figure 5: State Flow Diagram

The State flow diagram is what we made first. It indicates all operational states and transitions between them for a device. It acts as a clear guide using visual aids which help in understanding all features as well as sequence of actions within the system. We made sure that all eventualities and possible device behaviors were taken into account by painstakingly describing each state and its associated transitions. This method helps to debug and optimize the system as well as gives a strong base for further development phases. But in order to describe it let's see in detail:

The state flow diagram is divided into several key sections: (Note: The image always follows the explanation)

- **Start:** The start section is, as expected, the first part of the flow. When the program begins, it first activates all the necessary sensors and then enters the default mode, which is **Home Mode**.

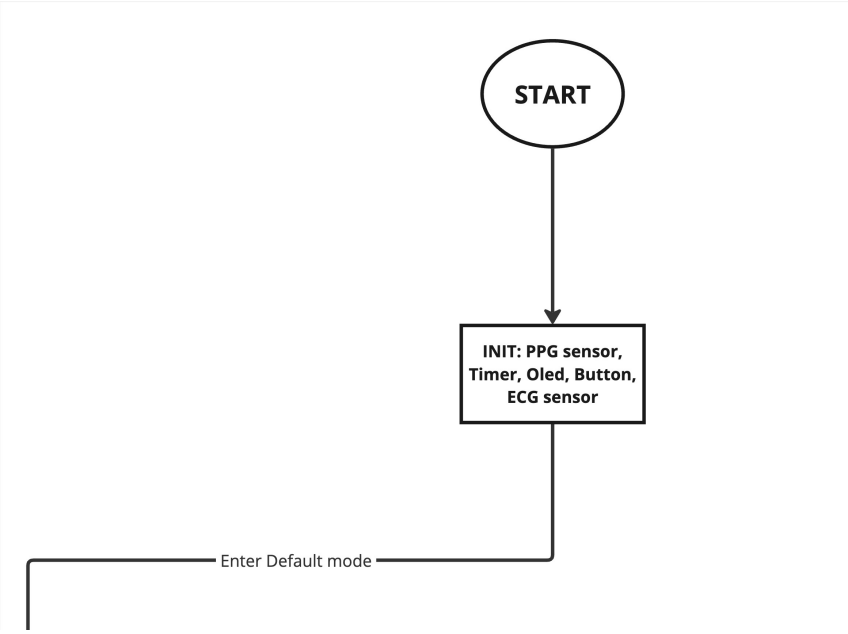



Figure 6: Start Section

- **Home Mode**: This is the first mode the system enters after initialization. In Home Mode, the system deactivates the ECG sensor and starts a timer. This mode serves as a default state where the system awaits further instructions.

  In this modality, we **chose** to use a PPG sensor to make the prototype (we'll discuss this in more detail in the next chapter), separate from the ECG sensor (which is external), and to take values useful to calculate **HR** and **HRV** from the wrist. Another reason was to ensure the opportunity to measure heart rate in a more accessible manner for everyone.

  1. **Timer and Contact Check**: After starting the timer, the system waits for 30 seconds and then checks if there is correct contact with the PPG sensor. If the contact is correct, the system proceeds with various tasks; if not, it shows messages indicating incorrect contact and turns on a green LED, **This last option has been introduced as projectual choice by us, in order to know when the sensor is correctly taking values also by visualizing the device**
  2. **Tasks upon Correct Contact**:
     - **Calculate HR and HRV:** The system calculates the heart rate (HR) and heart rate variability (HRV).
     - **Visualize HR and HRV on OLED:** The calculated HR and HRV values are displayed on an OLED screen.
     - **Visualize All Information on UART:** The system sends all collected information via UART for external processing or display.
     - **LED Intensity Adjustment:** The system adjusts the LED intensity to follow the heart rate, as per the project's design choice, **It was not requested to activate the led in the first screen, but we thought that was very useful to use it also in the Home Mode in order to visualize the pulse of the signal**. (It can obviously easy used only in the second screen by changing a simple flag in the code)
     - **Green LED Off:** The system turns off the green LED when the contact with the PPG sensor is correct. **This last option has been introduced as projectual choice by us, in order to know when the sensor is correctly taking values also by visualizing the device**

3. **High Heart Rate Check**: The system then checks if the heart rate is excessively high.
   - **If** HR is excessively high: The system shows alert messages on the display (a second heart) and sends these messages via UART to indicate the excessively high heart rate.
   - **If** HR is not excessively high: The system continues monitoring and processing as usual.
4. **Repeat Cycle**: This process repeats after every 30 seconds, ensuring continuous monitoring and real-time feedback based on sensor contact and heart rate data.

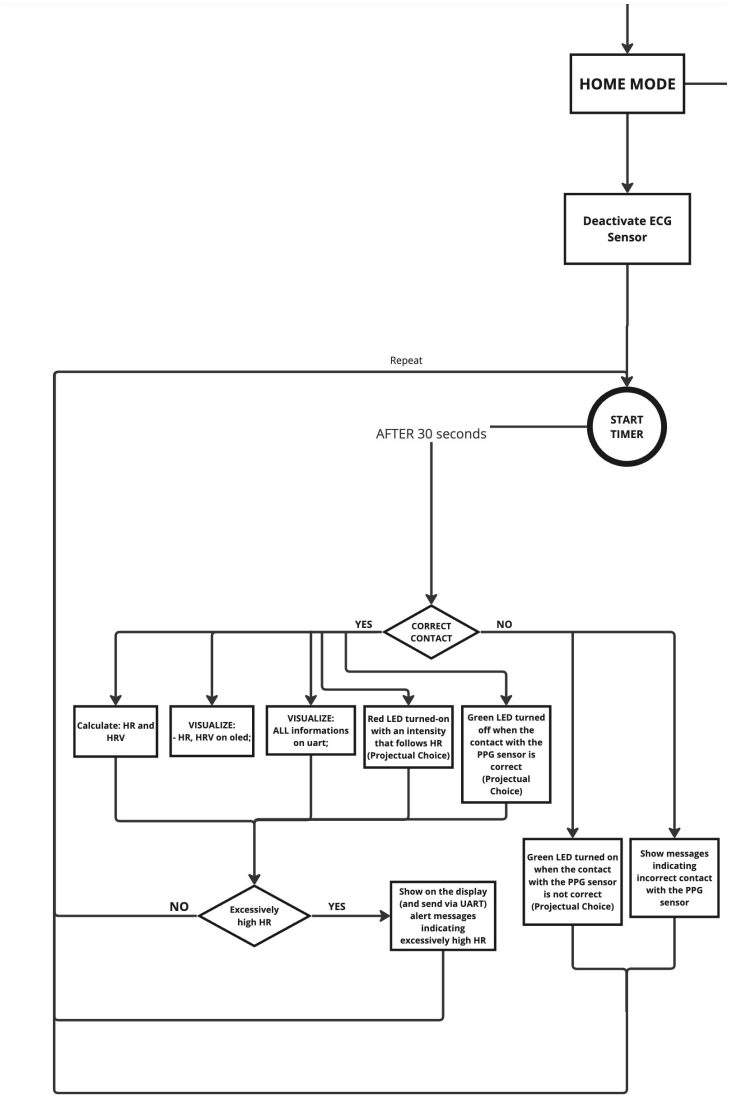



Figure 7: Home Mode

- **Check Button Click (Interrupt-Driven)**: This box represents a subprocess that is interrupt-driven, meaning it can be invoked at any point during the main process. When a button click is detected, the system immediately triggers the corresponding actions defined in the subprocess. **This is also a projectual representation choice, indicating that the button flow can be recalled at any moment during execution. Thanks to the interrupts, it is not constantly active, ensuring efficient operation.**

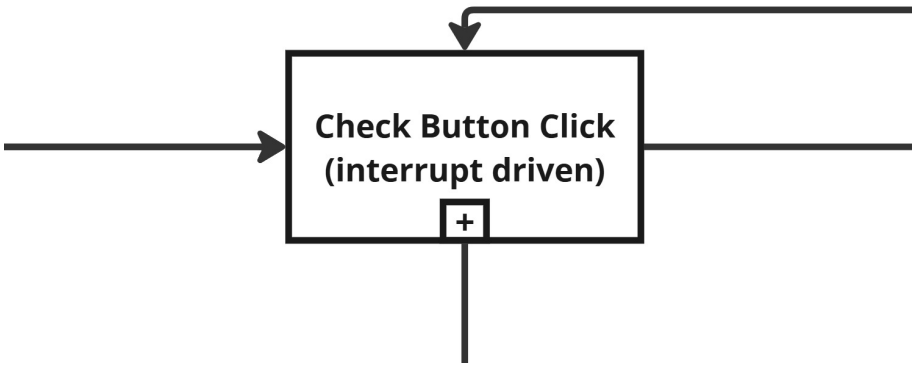



Figure 8: CheckButton Box interrupt driven

- **Button Click Subprocess**: The subprocess starts by checking the current mode of the system, which can be one of the following: Home Mode, Waveform Mode, or Advanced Mode. The system checks the modality in which we are in:
  1. **If we are in Home Mode: if** yes, it then checks for a button click, else it checks for other modalities
     - **If** a button click is detected, it changes the modality.
     - **If** no button click is detected, it continues monitoring.
  2. **If we are in Waveform Mode: if** yes, it then checks for a button click, else it checks for other modalities
     - **If** a button click is detected, it changes the modality.
     - **If** no button click is detected, it continues monitoring.
  3. **If we are in Advanced Mode: if** yes, it then checks for a button click, else it checks for other modalities
     - **If** a button click is detected, it changes the modality.
     - **If** no button click is detected, it continues monitoring.

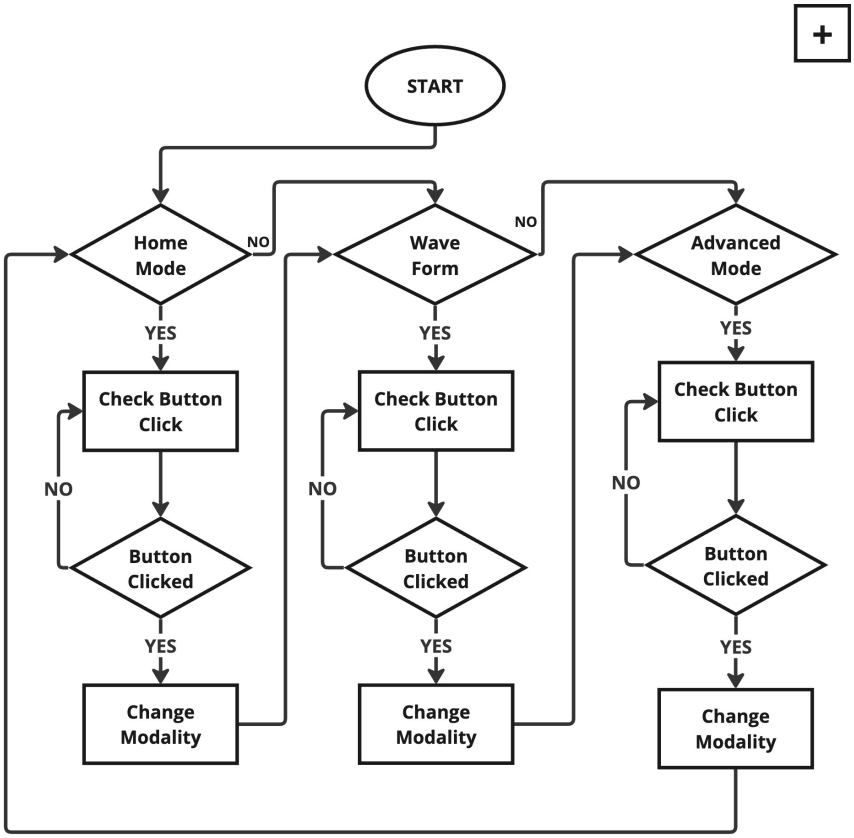



Figure 9: CheckButton Subprocess interrupt driven

- **Waveform mode:** In this and the following modality (advanced), the **HR** and **HRV** are calculated from the ECG sensor. This design choice was made to ensure the correct functioning of the program and because the ECG sensor, although more cumbersome and not suitable for the prototype, is more reliable.
  1. **Activate ECG Sensor:** Upon entering Waveform Mode, the system activates the ECG sensor to start collecting data.
  2. The system performs the following tasks (Note: the circular arrow on the boxes state that the task is repeated):
     - **Visualization of Waveforms:** The system visualizes both PPG and ECG waveforms on the OLED display in real-time.The displayed waveforms can optionally be filtered to eliminate noise.
     - **Send Raw Samples:** The raw samples acquired from the sensors are sent in real-time via UART for external processing or display. Formatting the data into two columns, one for each signal (ECG and PPG).

- **LED Intensity Adjustment:** The system adjusts the LED intensity to follow the heart rate.

3. **Correct Contact Check:** The system checks if there is correct contact with the PPG sensor.
   - **If the contact is correct**, The green LED is turned off.
   - **If the contact is incorrect**, The green LED is turned on and the system shows messages indicating incorrect contact with the PPG sensor and sends these messages via UART. **This part is also a projectual choice, meaning that it is not requested but we preferred to introduce it in order to have a visualizable part of the device**

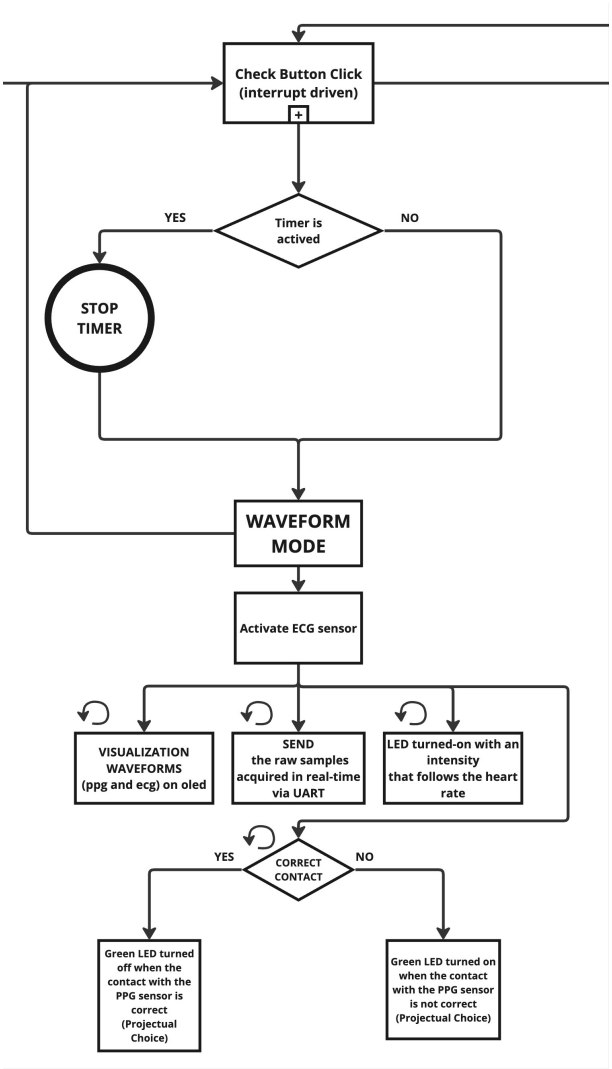



Figure 10: WaveForm Mode

- **Advanced Mode**:
  1. **Activate ECG Sensor:** Upon entering Advanced Mode, the system activates the ECG sensor to start collecting data. (We could also avoid specifying this activation because the sensor was already activated in the previous mode and not deactivated, as the modes are sequential. Despite this, we preferred to specify that the ECG sensor must be activated in this modality as well.)
  2. **The system performs the following tasks:**
     - **Extract Systolic/Diastolic Time Duration:** The system extracts the systolic and diastolic time duration from the PPG signal.
     - **Extract Pulse Arrival Time (PAT):** The system calculates the Pulse Arrival Time based on the collected data. ( i.e. the temporal delay between an R peak of the ECG and the corresponding systolic peak of the PPG signal)
     - **LED Intensity Adjustment:** The system adjusts the LED intensity to follow the heart rate, according to the project's design choice. **It was not requested to activate the led also in the third screen, but we thought that was very useful to use it also in the Advanced Mode in order to visualize the pulse of the signal**. (It can obviously easy used only in the second screen by changing a simple flag in the code)
     - **Correct Contact Check:** The system checks if there is correct contact with the PPG sensor.
       - **If the contact is correct**, The green LED is turned off.
       - **If the contact is incorrect**, The green LED is turned on and the system shows messages indicating incorrect contact with the PPG sensor and sends these messages via UART. **This part is also a projectual choice, meaning that it is not requested but we preferred to introduce it in order to have a visualizable part of the device**

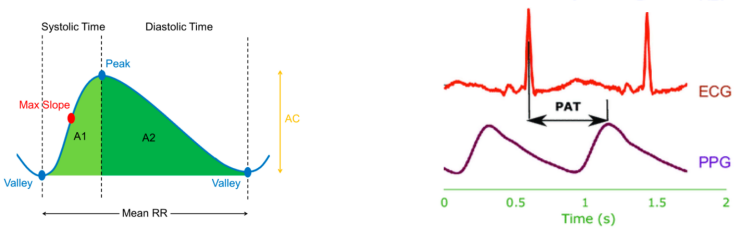



Figure 11: Graphical Representation of the Paarameters

3. **Start Timer:** The system starts a timer for continuous monitoring and data extraction. In particular The system graphically represents the temporal trends of heart rate (HR) and heart rate variability (HRV) based on the data collected over time. The Graphic is updated every 5 seconds. For **projectual reasons**, we chose to display only the HR graphical trend due to the restricted display size. However, the lines of code that show the HRV plot have been left inside the code and can be uncommented to visualize it.

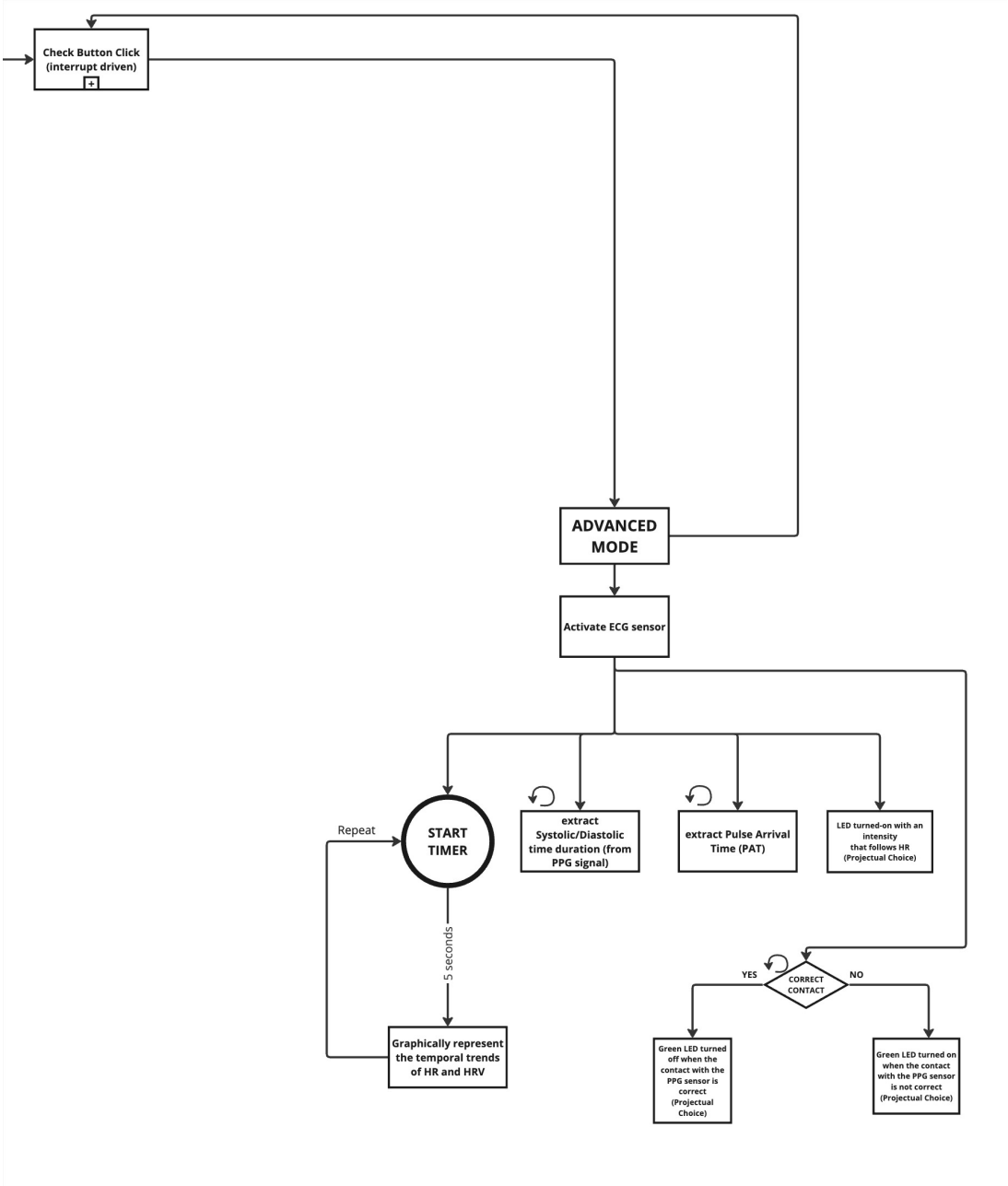



Figure 12: Advanced Mode

## 3.2 Available Sensors and instruments

Let's now introduce the set of sensors that we used in order to complete all the request of the project (we'll take a look each one in the next pages):

- 1 **STM-32 NUCLEO-G474RE**
- 1 **SSD1306 OLED display**
- 1 **PPG sensor**
- 1 **DS3231 external clock**
- 2 **external LED** one red and one green
- 1 **AD8232 ECG sensor**
- 1 **Button**

## 3.3 Protocols Involved by each sensor

- **SSD1306 OLED Display**: Uses the **I2C** protocol to communicate with the microcontroller, allowing efficient data transfer for display updates.
- **PPG Sensor**: Typically interfaces using an analog signal which is read by the microcontroller's ADC (Analog-to-Digital Converter)
- **DS3231 External Clock**: Uses the **I2C** protocol to communicate with the microcontroller, providing timekeeping data.
- **External LEDs (Red and Green)**: Controlled via **GPIO** (General-Purpose Input/Output) pins on the microcontroller.
- **AD8232 ECG Sensor**: Sends analog signals to the microcontroller's **ADC** for heart rate monitoring and processing.
- **Button**: Interfaced through GPIO pins, with interrupt capabilities to detect user inputs.

In addition to these protocols, we also used the **UART protocol** to send important information to the terminal and to plot values for visualizing the signal as requested.

## 3.4 Devices in detail

Now let's take a look to the devices in detail:

### 3.4.1 STM-32 NUCLEO-G474RE

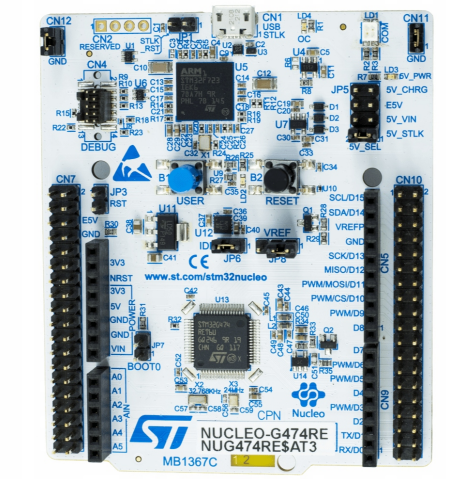



Figure 13: STM-32 NUCLEO-G474RE

The STM32 Nucleo-G474RE is a versatile development board based on the STM32G474RET6 microcontroller, part of the STM32 microcontroller family by STMicroelectronics. This board is designed to facilitate prototyping and the development of embedded applications.

**Main features**:

- **Microcontroller**: STM32G474RET6, based on a 32-bit ARM Cortex-M4 core, with floating-point unit (FPU) and DSP support.
- **Clock frequency**: Up to 170 MHz.
- **Memory**: 512 KB of flash memory and 128 KB of SRAM.
- **Interfaces**: Includes a variety of communication interfaces such as USART, I2C, SPI, CAN, USB, and others.
- **Connectivity**: Connectors compatible with Arduino Uno V3 and ST morpho, allowing expansion with a wide range of shields and modules.
- **Debug and programming**: Integrated ST-LINK/V2-1, offering on-board debug and programming capabilities.

### 3.4.2 SSD1306 OLED Display

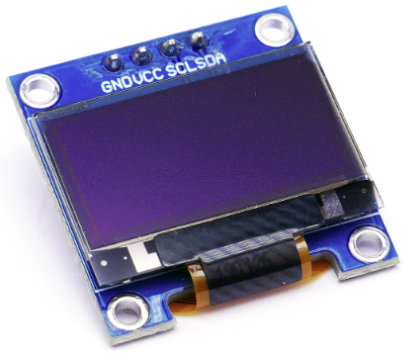



Figure 14: SSD1306

The SSD1306 is a compact, high-resolution OLED (Organic Light Emitting Diode) display that provides clear and bright visual output. This display is particularly suitable for wearable and portable devices due to its low power consumption and high contrast ratio, which ensures readability even in various lighting conditions.

**Functionality and Features**:

- **Resolution and Size:** The SSD1306 display typically comes in a 128x64 pixel resolution, which provides ample space for displaying text, graphs, and other graphical elements.
- **Communication Protocols:** It communicates with the microcontroller via I2C . This protocols are chosen for their simplicity and efficiency in data transfer, enabling quick updates to the display.
- **Power Consumption:** One of the key advantages of the OLED technology is its low power consumption. Unlike traditional displays, OLEDs do not require a backlight, as each pixel emits its own light. This not only reduces power usage but also allows for deeper blacks and higher contrast.
- **Driver and Control:** The SSD1306 has an integrated controller that handles the display logic. It interprets commands and data sent from the microcontroller to control the display's pixels. The microcontroller sends commands to initialize the display, set the display parameters, and update the content. In particulare, in order to use it into the IDE we had to create some drivers.
- **Graphics and Text Display:** The display can render both graphics and text. Libraries such as Adafruit's SSD1306 library for Arduino or similar libraries for

STM32 can be used to simplify the control and rendering of images, shapes, and text on the display. These libraries provide functions to draw lines, circles, rectangles, and to display fonts of various sizes.

- **Refresh Rate:** The SSD1306 display supports a refresh rate that is sufficient for displaying dynamic content, such as scrolling text or changing graphs. This makes it ideal for real-time monitoring applications where data is frequently updated.

**Usage in the Project:** In our project, the SSD1306 OLED display is used to show real-time data from various sensors, including heart rate, heart rate variability (HRV), and other monitored parameters. The display serves as the main user interface, providing clear and immediate feedback to the user. It shows screens, alerts, and other interactive elements. The OLED display can graphically represent trends and changes in the collected data, such as plotting heart rate over time or showing the waveform from the ECG sensor. This visual representation helps users quickly understand their health metrics.

By integrating the SSD1306 OLED display into our system, we ensure that users have access to critical information at a glance, presented in a clear and intuitive manner. The combination of its advanced features and ease of use makes it an essential component of our health monitoring project.

### 3.4.3 PPG Sensor

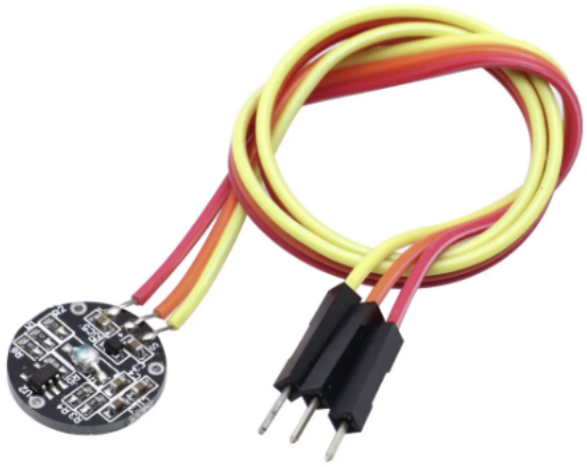


Figure 15: PPG sensor

The **light-tissue interaction** mechanism allows to obtain signals related to vital signs and one of these signals is the photopletysmogram (PPG). The PPG sensor is designed to detect blood volume changes in the microvascular bed of tissue, which helps in measuring heart rate and other cardiovascular parameters. This non-invasive sensor is crucial for continuous health monitoring and provides valuable insights into the user's physiological state. The device based on PPG signal is known as **pulse oximeter**

**Functionality and Features**:

- **Detection Principle:** The PPG sensor works by emitting light into the skin and measuring the amount of light either transmitted or reflected to detect blood volume changes. The sensor we have typically uses infrared or green light for this purpose, as these wavelengths penetrate the skin effectively.
- **Communication Protocol:** The sensor can interface with the microcontroller using analog signals, which are read by the microcontroller's ADC (Analog-to-Digital Converter).
- **Sampling Rate:** The sensor operates at a sampling rate that is adequate for capturing the pulse waveforms accurately, allowing precise heart rate calculations and analysis of heart rate variability (HRV).
- **Signal Processing:** The sensor's raw data often requires signal processing to remove noise and artifacts. This is typically handled by the microcontroller or a

dedicated signal processing unit to ensure accurate heart rate and HRV measurements.

**Usage in the Project**:

**Heart Rate Monitoring:** The PPG is primarily used to measure the heart rate by detecting the blood volume changes with each heartbeat. This data is crucial for tracking the user's cardiovascular health in real time.

**Heart Rate Variability (HRV):** By analyzing the time intervals between heartbeats, the sensor helps in calculating HRV, providing insights into the autonomic nervous system and overall health.

**Graphical Display:** The processed data from the PPG is displayed on the SSD1306 OLED, allowing users to visualize their heart rate trends and other related metrics in real time. Integrating the PPG sensor into our health monitoring system enhances its capability to provide accurate and continuous cardiovascular monitoring, offering users valuable health insights with minimal intrusion.

### 3.4.4 DS3231 external clock

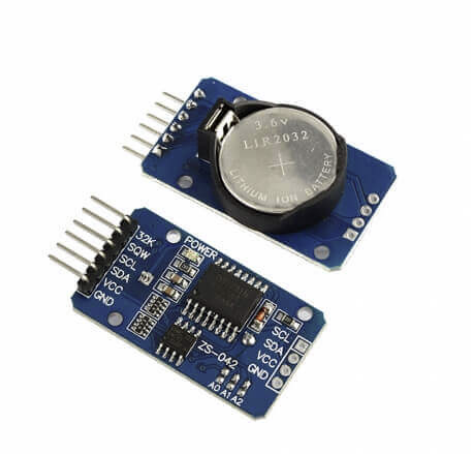



Figure 16: DS3231 external clock

The **DS3231** is a high-precision real-time clock (RTC) module that is essential for keeping accurate time and date information in various applications, including health monitoring systems. This module ensures that all data collected is accurately timestamped, which is crucial for tracking trends and analyzing long-term health data.

**Functionality and Features**:

- **High Accuracy:** The DS3231 is known for its high accuracy due to the integrated temperature-compensated crystal oscillator (TCXO). It maintains precise timekeeping with an accuracy of ±2 minutes per year, even under varying environmental conditions.
- **Timekeeping:** The DS3231 keeps track of seconds, minutes, hours, day, date, month, and year with leap-year compensation valid up to 2100. This ensures that the system maintains accurate timekeeping without the need for frequent manual adjustments.
- **Communication Protocol:** It communicates with the microcontroller via the I2C protocol. This two-wire communication method is both simple and efficient, allowing easy integration with the microcontroller and other I2C-compatible devices.
- **Battery Backup:** One of the key features of the DS3231 is its ability to operate on a backup battery when the main power is off. This ensures that the clock continues to keep accurate time even during power outages or when the device is turned off.

- **Low Power Consumption:** The DS3231 is designed to consume minimal power, which is especially important for battery-operated and portable devices. The low power consumption ensures that the backup battery can last for years.

**Usage in the Project:**

**Accurate Timestamping:** The primary use of the DS3231 in our project is to provide accurate timestamps for the data collected by various sensors and send them via **UART** to the terminal. This allows for precise tracking of changes and trends over time, which is crucial for effective health monitoring.

Integrating the DS3231 external clock into our health monitoring system provides a reliable and accurate timekeeping solution that enhances the overall functionality and resilience of the device. Its features ensure that all data is accurately timestamped and that the system can perform scheduled tasks reliably, making it an indispensable component of our project. In particular its usage has been fundamental in order to achieve the correct moment (data and hour) of the measure.

### 3.4.5 External LEDs (Red and Green)

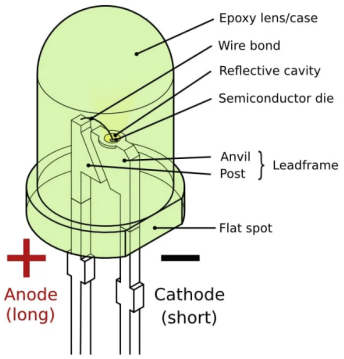



Figure 17: Led Scheme

Figure 18: Led used for the project

**Light Emitting Diodes (LEDs)** are an essential component in modern electronics, known for their ability to emit light when an electric current passes through them. They belong to a broader category of devices called diodes, which are two-terminal components designed to allow current to flow primarily in one direction. This characteristic of diodes, known as asymmetric conductance, makes them invaluable in controlling the direction of current in circuits.

LEDs distinguish themselves from other diodes through their light-emitting capability. This phenomenon occurs within the semiconductor material of the LED. When the LED is forward biased (meaning the positive voltage is applied to the anode and the negative to the cathode) the potential barrier within the diode decreases. This reduction allows electrons to move across the junction and recombine with holes (the absence of an electron) in the depletion layer. During this recombination process, energy is released in the form of photons, which we perceive as light. The specific color of this light is determined by the energy band gap of the semiconductor material used in the LED.

LEDs are characterized by several key parameters:

- **Current (I)**: The typical operating current for LEDs ranges from 0.5 mA to 40 mA. This current is the amount of electric charge flowing through the LED.
- **Voltage (V)**: The voltage required to operate an LED generally falls between 1.1 V and 4 V, depending on the type and color of the LED.

- **Power (W)**: The power consumed by an LED is the product of its voltage and current. For example, a red LED might operate at around 0.016 W.

Figure 19: Resistor

When integrating LEDs into electronic circuits, it is crucial to consider their sensitivity to current. LEDs can be easily damaged by excessive current, leading to permanent failure. Therefore, they must be protected by a current-limiting device, such as a resistor. A ballast resistor, specifically, is used in LED circuits to limit the current flow to safe levels. For instance, a standard 5 mm LED typically requires a maximum current of 20 mA. Without a ballast resistor, connecting such an LED directly to a power source could result in a current that exceeds this safe limit, causing the LED to burn out almost instantly.

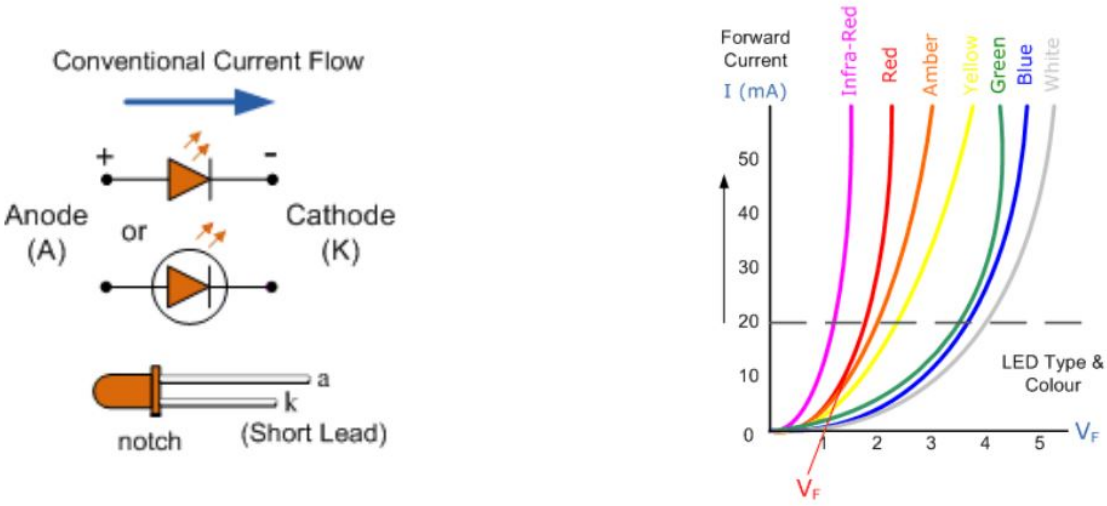



Figure 20: Led Type and Colour

**Usage in the Project**:

LEDs are used to provide visual feedback based on signals from the sensors, enhancing both functionality and user experience. Two types of LEDs, a red LED and a green LED, are implemented, each serving distinct roles.

The red LED indicates the heartbeat detected by a sensor. It follows the intensity of the heartbeat, providing a visual representation of heart activity. Using Pulse Width Modulation (PWM), the red LED's brightness gradually increases and decreases, making the heart rate more discernible and visually engaging. This modulation allows users

to better observe the heart's rhythm and intensity.
The green LED was introduced by us as **projectual choice** (it was not requested) and serves as an indicator for the PPG sensor's status. It lights up when the PPG sensor does not detect a proper signal, typically due to poor finger contact. This visual cue helps users ensure the sensor is correctly positioned, thereby ensuring accurate readings.

(Note: All the schemes and hardware projectual choices about the LEDs will be explained better in the next chapters).

### 3.4.6 AD8232 ECG Sensor

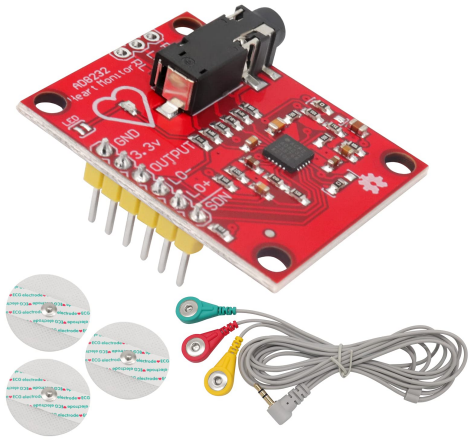



Figure 21: AD8232 ECG Sensor

The **AD8232** ECG sensor is a commercial board designed to measure the electrical activity of the human heart, producing an analog signal that can be displayed as an electrocardiogram (ECG). This sensor is particularly adept at reducing noise, a common issue in ECG signals, using its integrated AD8232 chip. The chip functions similarly to an operational amplifier, helping to provide a clear signal from the typically noisy intervals of biopotential measurements.

The primary role of the sensor is to extract, amplify, and filter small biopotential signals, even under noisy conditions such as those caused by motion or remote electrode placement. This makes it suitable for various ECG and biopotential measurement applications. The chip can implement a two-pole high-pass filter to eliminate motion artifacts and electrode half-cell potentials, tightly coupled with the amplifier's architecture. This allows both significant gain and high-pass filtering in a single stage, saving space and cost. Additionally, an uncommitted operational amplifier in the AD8232 enables the creation of a three-pole low-pass filter to remove additional noise, with user-selectable frequency cutoffs to accommodate different applications.

For enhanced noise rejection, the AD8232 includes an amplifier designed for driven lead applications, such as right leg drive (RLD). It also features a quick restore function, which reduces the duration of long settling tails in the high-pass filters, allowing the sensor to recover rapidly from abrupt signal changes like leads-off conditions. This

ensures that valid measurements can be taken shortly after reconnecting the electrodes.

**Pin Configuration**:

- **SDN**: Used to shutdown the sensor and connected to a GPIO output pin
- **LO+**, **LO-**, Used detect the correct contact of the sensor and connected to GPIO Input
- **OUTPUT**, **3.3V**, and **GND** Pins: These pins allow for easy integration with development boards.
- **Right Arm (RA), Left Arm (LA), and Right Leg (RL) Pins**: For connecting custom sensors.
- **LED Indicator**: Displays the heartbeat rhythm.

Operating on a single supply voltage ranging from 2 V to 3.5 V, the AD8232 is designed for low power consumption, drawing only 170 µA. It also features an internal RFI filter. Packaged in a compact 4 mm × 4 mm, 20-lead LFCSP, it operates over a wide temperature range from $-40°C$ to $+85°C$, with specified performance from $0°C$ to $70°C$.

**Usage in the Project**:

The AD8232 sensor is a vital component in our project, specifically used in the last two modalities. This sensor plays a crucial role in providing accurate and reliable biopotential signals.
The sensor is designed to detect the electrical activity of the heart, providing real-time data on heart rate. In our project, this sensor is used to aquire the heart's electrical signals, even in noisy conditions. This ensures that the heart rate data is accurate and consistent, which is critical for effective health monitoring.
The processed datas are displayed on the SSD1306 OLED screen. This allows users to visualize their heart rate trends and other related metrics in real-time. By integrating it into our health monitoring system, we enhance its capability to provide continuous and precise cardiovascular monitoring.

### 3.4.7 Button

A **push button** is a simple switch mechanism used to either allow or interrupt the flow of electrical current when pressed. In its default state, when the button is not pressed, it is considered "open," meaning it does not connect to any voltage level and is thus a floating point. To ensure a defined logic state (HIGH or LOW) when the button is not pressed, a pull-up or pull-down resistor is used

Pull-up resistors connect the input pin of a microcontroller to the power supply voltage (Vcc), ensuring a default HIGH state when the button is open. Conversely, pull-down resistors connect the input pin to ground (0 V), resulting in a default LOW state when the button is open. These resistors are crucial for ensuring that the digital input pin on a microcontroller has a defined state, preventing it from floating and potentially causing erratic behavior.
When the push button is pressed, the electrical contacts inside the switch come into contact, allowing current to flow. However, these contacts do not form a clean, low impedance path immediately. Instead, they tend to rub and bounce against each other several times before settling, causing multiple transitions between open and closed states. This phenomenon, known as "**bouncing**," can cause digital circuits to misinterpret these multiple transitions as several button presses. To mitigate this issue, debouncing techniques are employed. These can be implemented in both hardware and software.

**Usage in the Project**:
The push button in our project serves as a crucial interface for mode selection, allowing users to switch between different operational modes with ease. It provides a simple yet effective means of navigating through the various functionalities of the system.
The button enables the user to cycle through the three distinct modes: Home mode, Waveform mode, and Advanced mode.

- **First Press**: When the push button is pressed once, the system switches from Home mode to Waveform mode.
- **Second Press**: Pressing the button again changes the mode from Waveform mode to Advanced mode.
- **Third Press**: Pressing the button once more returns the system to Home mode.

## 3.5 Development Environment and Board Configuration

To program the NUCLEO-G474RE board, we opted for the STM32CubeIDE programming environment. This choice was driven by STM32CubeIDE's full compatibility with our board and our familiarity with the environment, having used it extensively throughout our coursework. The IDE offers a range of debugging tools and utilities that simplify the modification of the board's configuration files, making it an ideal choice for our project.

### 3.5.1 Configuration

We utilized STM32CubeMX within STM32CubeIDE to set up the .ioc file, which is essential for the operation of our board. STM32CubeMX allows us to configure all environment parameters and specify the behavior for each implemented protocol, including the operational parameters of the board and the configuration of its physical connections. After evaluating the sensors and actuators required for our project, we identified the need to implement the following technologies:

- **I2C**
- **UART**
- **ADC**
- **TIM**
- **TIM with PWM**

After careful consideration and analysis of our requirement to measure vital parameters over periods of several seconds, we determined that a high clock speed is necessary to ensure precise and timely data processing. Consequently, we selected the highest clock frequency offered by our board: 170 MHz.

This high clock speed allows for rapid data acquisition and processing, which is crucial for real-time monitoring of physiological signals. With a 170 MHz clock, the microcontroller can efficiently handle the simultaneous tasks of reading sensor data, processing signals, and updating the display without lag. This ensures that the system remains responsive and accurate, providing reliable health metrics to the user.

In addition, the high clock speed supports advanced signal processing algorithms that are essential for deriving meaningful insights from raw sensor data. These algorithms

can perform complex calculations such as heart rate variability (HRV) analysis, noise filtering, and artifact detection, which enhance the accuracy and reliability of the measurements.

By leveraging the full potential of our microcontroller's clock speed, we ensure that our health monitoring system meets the stringent performance requirements necessary for continuous and accurate vital parameter measurement.

### 3.5.2 I2C

The Inter-Integrated Circuit (I2C) is a hardware specification and protocol developed by Philips. The main characteristics are the following.

- There are only two wires: one for data and one for clock.
- It is synchronous half-duplex, with a clock line.
- Single-ended 8-bit oriented serial bus.
- Open-collector/Open-drain bus (TTL level).
- Multi-slave.
- Devices can communicate at 100kHz (*standard mode*) or 400kHz (*fast mode*). Recent revisions of the standard can run at faster speeds (1MHz, known as fast mode plus, 3.4MHz, known as high speed mode, and 5MHz, known as ultra fast mode). This is important to know for the project because we used this new feature in order to send informations to the SSD1306

The two wires forming an I2C bus are bidirectional open-drain lines, named **Serial Data Line** (SDA) and **Serial Clock Line** (SCL). The protocol specifies that these two lines need to be pulled up with resistors. The sizing of these resistors depends on the bus capacitance. It is quite common to use resistors with a value close to $4.7K\Omega$. When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. All the devices need to pull the bus to a logical level of 1 to represent a logical high.

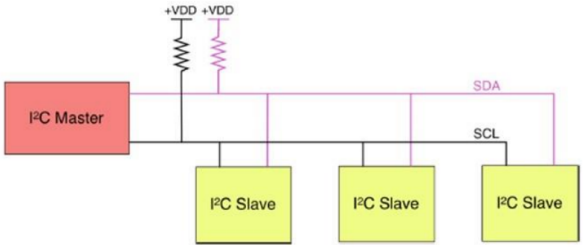
The peripherals used in our project for the I2C bus are two of those mentioned before:

- **SSD1306**: OLED Display
- **DS3231**: External Clock

We utilized a single I2C bus, specifically the I2C3, to communicate with these devices.

Using a single I2C bus offers several advantages:

- **Simplified Wiring**: A single I2C bus reduces the complexity of the wiring, as multiple devices can share the same two-wire bus (SDA and SCL). This simplifies the design and assembly of the circuit.
- **Efficient Use of Microcontroller Pins**: By sharing the I2C bus, we minimize the number of pins required on the microcontroller, freeing up additional pins for other peripherals and functions.




- **Scalability**: The I2C protocol supports multiple devices on the same bus, identified by unique addresses. This makes it easy to add more I2C-compatible devices to the system in the future without needing additional communication lines.
- **Synchronization and Coordination**: The I2C protocol inherently manages the synchronization of data transfer between the master (microcontroller) and the slave devices (SSD1306 and DS3231), ensuring reliable communication even when multiple devices are active on the bus.
- **Cost-Effective**: Reducing the number of communication lines and simplifying the circuit design can lead to cost savings in terms of both components and manufacturing.

In our project, the use of I2C3 for both the SSD1306 OLED display and the DS3231 external clock ensures efficient and reliable data communication. The I2C bus allows the microcontroller to seamlessly manage the display updates and real-time clock synchronization without interference or data collision. This integration enhances the overall functionality and performance of our health monitoring system.

### 3.5.3 TIM

A **timer** is a free-running counter with a counting frequency that is a fraction of its clock source. Usually, a timer counts from zero up to a given **Period** value, but it can also count on the contrary and in other ways. They can be used to control an output

waveform, or to indicate when a period of time has elapsed (output compare mode). They can be used to generate **PWM signals** or to measure the frequency of an external event (input capture mode).
A timer can receive signals from other sources. These can be divided in two main groups:

- **Clock sources**: A timer cannot work without a clock source, because this is used to increment the counter register. The timer can be clocked by the *internal clock* (which is derived from the bus where it is connected) or by an *external clock* source. A timer can be also clocked by another timer used as "master".
- **Trigger sources**: A timer can be configured to start counting when an external event triggers it. This is used to synchronize the timer with external sources connected to the MCU pins or with other timers connected internally.

Depending on its type, a timer can generate interrupts or DMA requests.

Timers were crucial in our project for synchronizing the fading of the LED with the acquired signal (PWM), as well as for managing the signal acquisition. In our case, timers were used in various modes: polling mode (**TIM2** for **PWM**), interrupt mode (**TIM3** for performing actions after a set period) and to configure the DMA for ADC (**TIM2**). They were essential for several reasons:

- **Synchronizing LED Fading** with Acquired Signals (PWM): We used timers to generate PWM signals that control the intensity of LEDs. This allowed us to synchronize the fading of the LEDs with the acquired heart rate signals, providing a visual representation of the heartbeats.
- **Managing Signal Acquisition:** Timers were used to manage the acquisition of signals from various sensors. By setting precise intervals, we ensured that data was sampled consistently and accurately, which is critical for reliable measurements and analysis
- **Efficient Data Processing:** The use of timers enabled efficient data processing. By triggering events at specific intervals, we could process and analyze the data in real-time, ensuring that the system responded promptly to changes in the sensor readings.
- **Reducing CPU Load:** Timers offload the task of timekeeping from the CPU, allowing it to perform other operations. This is especially important in real-time systems where timely responses are crucial.

### 3.5.4 UART

A **Universal Synchronous (or asynchronous) Receiver/Transmitter (UART/USART)** is a device that translates a parallel sequence of bits (usually grouped in a byte) in a continuous stream of signals flowing on a single wire.

- A Universal Synchronous Receiver/Transmitter (USART) interface is a device able to transmit data word serially using two I/Os:
  - one acting as **transmitter** (TX)
  - one as **receiver** (RX)
  - **an additional I/O as one clock line**
- A Universal Asynchronous Receiver/Transmitter (UART) uses only two RX/TX I/Os

**USART** is particularly useful for applications requiring synchronization between the transmitting and receiving devices. The clock line ensures that data is transmitted and received at the same rate, reducing errors in communication.
In our project, the UART/USART interface is used for several key functions:

- **Data Transmission to External Devices**: We use the UART interface to send collected sensor data to external devices, such as computers or additional processing units. This allows for further analysis, logging, and visualization of the data.
- **Debugging and Testing**: During development, the UART interface is invaluable for debugging and testing. By sending diagnostic messages and system statuses through UART, we can monitor the internal states of the system in real-time and troubleshoot issues effectively.

### 3.5.5 ADC

Least but not last, we used ADC in order to acquire signals from the sensors, we have in dotation.

Analog-to-Digital Converters (ADCs) are crucial electronic devices that transform continuous analog signals into a digital form that computers can process. These devices are essential in many modern applications, including telecommunications systems, electronic measurement instruments, and consumer electronic devices.
The basic function of an ADC involves sampling the input analog signal at a consistent

rate and then quantizing the sampled values into a digital format. The performance of an ADC is characterized by:

- **Resolution:** This is the smallest change in the analog input signal that results in a change in the digital output. It is typically expressed in bits. For instance, a 12-bit ADC can represent an analog input with one of 4096 different levels.
- **Sampling Rate:** This is the frequency at which the ADC samples the analog input. It determines how frequently the analog signal is updated to a digital value and is usually measured in samples per second (sps).

In our project, the ADCs were essential for several key functions:

- **Signal Acquisition**: We used ADCs to convert the analog signals from sensors such as the PPG and ECG sensors into digital form. This conversion is crucial for the microcontroller to process and analyze the data accurately.
- **High Resolution for Precision**: The high resolution of our ADCs ensures that even small changes in the sensor signals are captured accurately. This is particularly important for physiological measurements, where precision is critical for reliable monitoring and analysis.
- **Real-Time Data Processing**: With a high sampling rate, the ADCs allow us to capture rapid changes in the sensor signals, enabling real-time data processing and analysis. This capability is essential for applications such as heart rate monitoring, where timely responses to changes in physiological parameters are crucial.
- **Efficient Use of DMA**: To handle the continuous stream of data from the sensors efficiently, we used DMA (Direct Memory Access) in conjunction with our ADCs. This setup allows the ADC to transfer data directly to memory without burdening the CPU, ensuring smooth and efficient data acquisition.

**Note**: In these paragraphs, we did not delve deeply into the details of every part of the code. We will explore these aspects more thoroughly in the following chapters.

## 3.6 .ioc: Configurations

The STM32 Cube IDE provides the capability to view detailed pinout configurations. Additionally, the devices connected to all peripherals can be renamed and configured in various ways to meet our specific requirements.

This flexibility is particularly useful for our project because it allows for precise customization and optimization of the hardware setup. For instance, we can easily assign specific pins to different functions, such as GPIO, I2C, UART, and PWM, ensuring that all components are connected efficiently.

**Pin Configuration**:

- The pinout view in STM32 Cube IDE helps in visualizing which pins are being used for each peripheral and ensures there are no conflicts.
- Renaming pins and peripherals in the IDE improves readability and makes the project easier to manage, especially as it grows in complexity.
- The IDE allows for quick changes to pin configurations, which can be essential during the development and testing phases.

### 3.6.1 PinOut Configuration

The following image provides a summary of all the pins used and renamed. For the sake of clarity, all names are listed below:

- PA0: LED
- PA1: Led_Sensor
- PA2: USART2_TX
- PA3: USART2_RX
- PA10: Lo_Minus
- PC2: PPG
- PC3: ECG
- PC4: Lo_Plus
- PC5: SDN
- PC8: I2C3_SDA
- PC9: I2C3_SCL
- PC13: Button

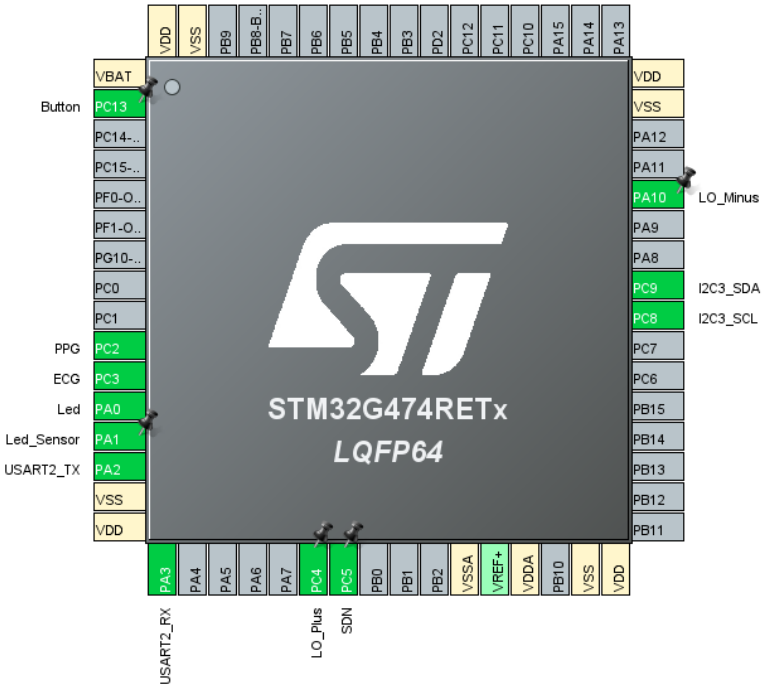



Figure 22: PinOut Configuration

### 3.6.2 Clock Configuration

Here's the schematic of the Clock configuration of the entire device. The clock configuration is crucial for ensuring that all peripherals operate correctly and in synchronization.

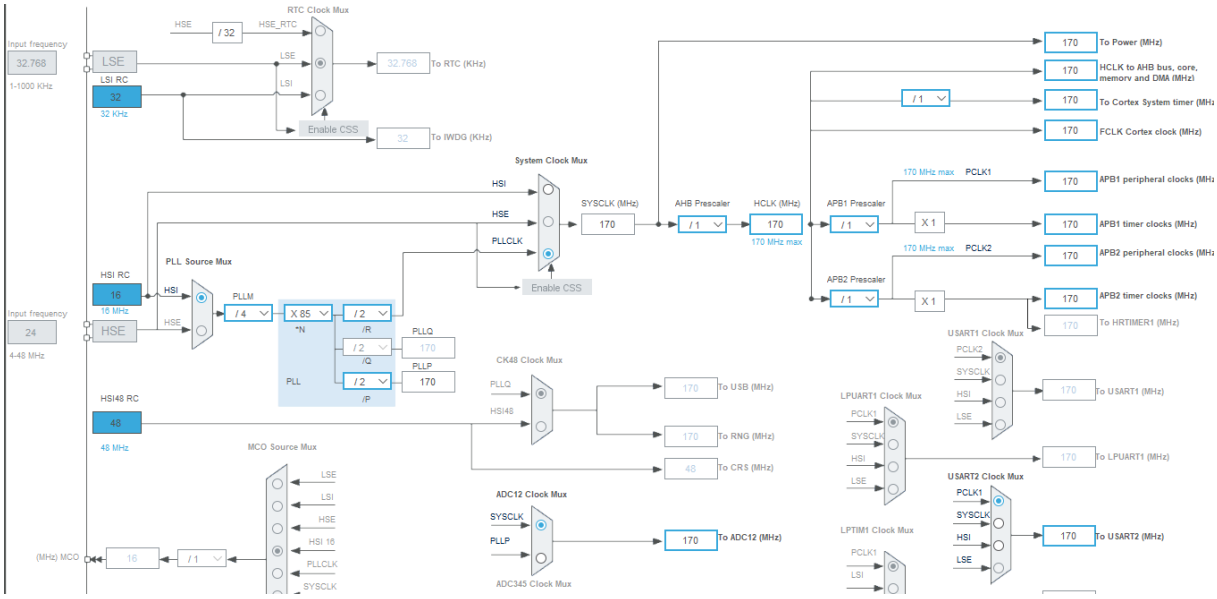



Figure 23: Clock first page

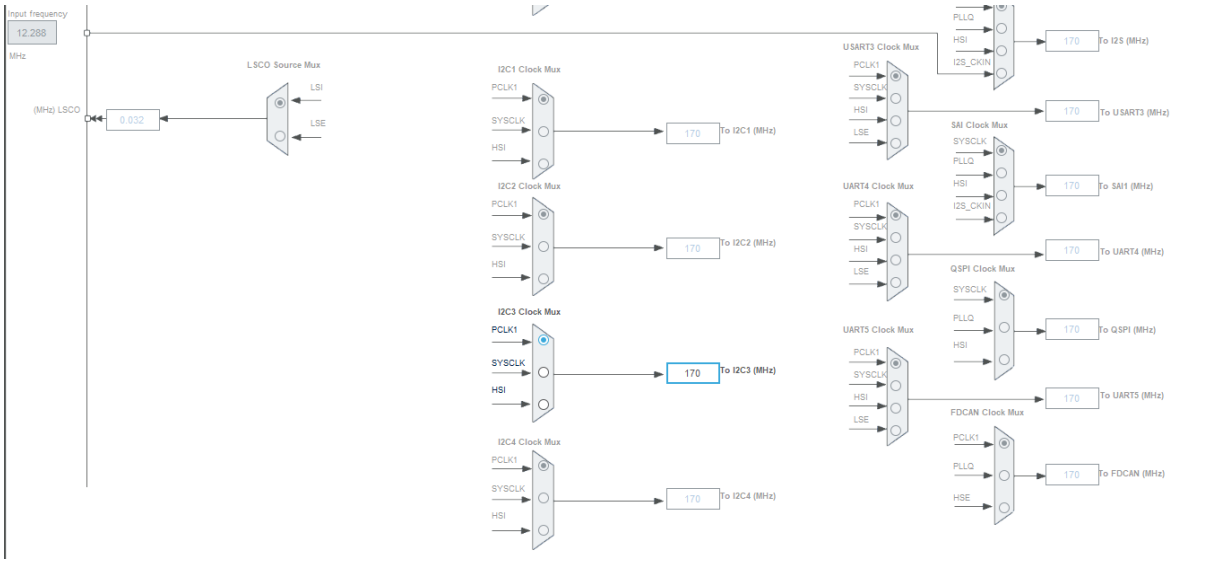



Figure 24: Clock second page

### 3.6.3 Protocols Configuration:

#### ADC Configuration Details

The configuration of the ADC (Analog-to-Digital Converter) was critical to ensure accurate and efficient data acquisition from our sensors. Below are the detailed settings used for our ADC (Note that only most significant choices for the project will be explained):

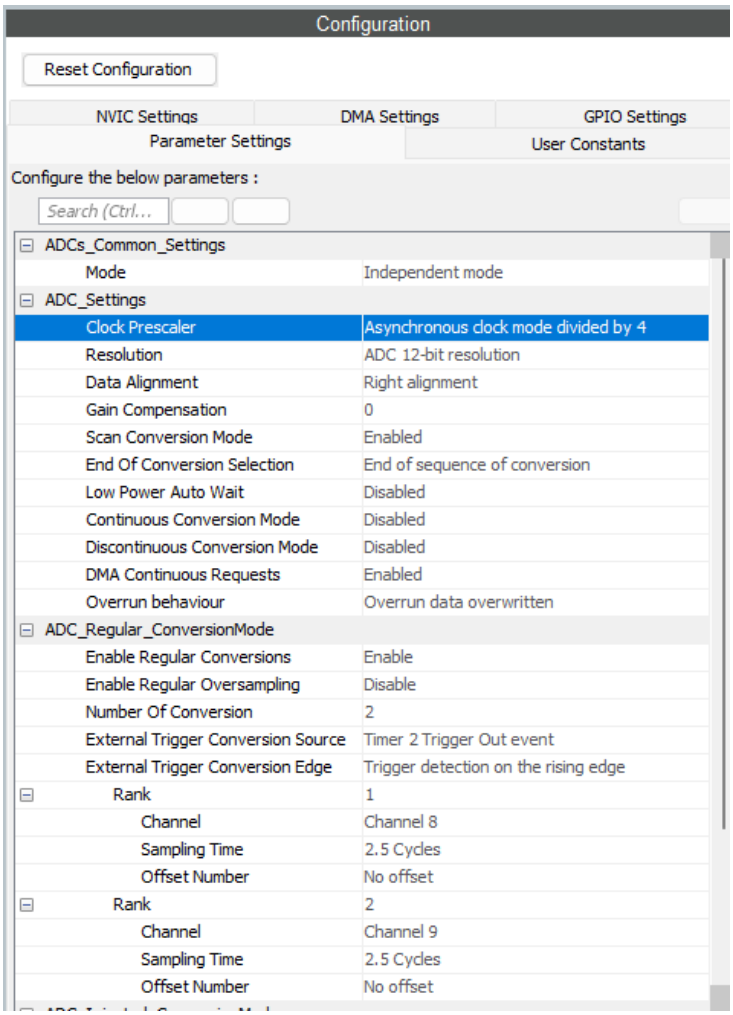



Figure 25: ADC Configuration

- **Clock Prescaler**: Asynchronous clock mode divided by 4
  - The ADC clock is derived from an asynchronous clock source and divided by a factor of 4. This reduction helps to minimize noise and enhance the accuracy of conversions.
  - **Avantage:** Using an **asynchronous clock** allows for independent control over the ADC's sampling rate, which can be crucial for optimizing performance

in terms of noise reduction and power efficiency. During our data acquisition we estabilished that this modality was the best compromise in order to acquire datas with no high noises and no very slow data acquisition.

- **Resolution**: ADC 12-bit resolution
  - The ADC converts analog signals into digital values with a 12-bit resolution, meaning the digital output can have 4096 distinct values (from 0 to 4095).
  - **Advantage**: Higher resolution provides more precise measurements of the analog signals, which is essential for applications requiring fine granularity in data acquisition.
- **DMA Continuous Requests**: Enabled
  - Continuous DMA requests are enabled, allowing the DMA controller to continuously transfer data from the ADC to memory *without* CPU intervention,. This allowed to acquire datas in **real-time**, otherwise it was not possible to do it.
  - **Advantage**: Using DMA reduces the CPU load and increases the efficiency of data transfer, allowing the microcontroller to handle other tasks simultaneously.
- **Enable Regular Conversions**: Enabled
  - Regular ADC conversions are enabled, ensuring continuous data acquisition.
  - **Advantage**: Continuous data acquisition is essential for real-time monitoring and processing of sensor data.
- **Number Of Conversions**: 2
  - The ADC performs two conversions in the sequence, capturing data from two different channels.
  - **Advantage**: This allows the system to monitor and process multiple signals simultaneously, enhancing the functionality of the device.

- **External Trigger Conversion Source**: Timer 2 Trigger Out event
  - The ADC conversion sequence is triggered by the Timer 2 Trigger Out event, ensuring synchronized data collection with other system events.
  - **Advantage**: Synchronizing ADC conversions with timer events helps in maintaining a consistent and predictable data acquisition rate. (In TIM2 configuration will be better explained how timer and ADC works together)
- **External Trigger Conversion Edge**: Trigger detection on the rising edge
  - The ADC conversion is initiated on the rising edge of the external trigger, providing precise control over the timing of data acquisition.
  - **Advantage**: Precise control over timing improves the reliability and accuracy of the data collected from the sensors.
- **Channel Configuration**:
  - **Rank 1**:
    - **Channel**: Channel 8
    - **Sampling Time**: 6.5 Cycles
    - **Offset Number**: No offset
    - **Advantage**: Short sampling time ensures rapid data acquisition, which is beneficial for high-speed applications, and in this case was crucial that it was very fast, in order to acquire datas with an high velocity.
  - **Rank 2**:
    - **Channel**: Channel 9
    - **Sampling Time**: 6.5 Cycles
    - **Offset Number**: No offset
    - **Advantage**: Similar to Rank 1, a short sampling time for Channel 9 ensures quick and efficient data acquisition from multiple sensors.

### Timers Configuration Details: TIM2

The configuration of TIM2 was crucial for generating PWM signals and ensuring precise timing for various operations. Below are the detailed settings used for TIM2 (Note that also in this case, just crucial parameters will be explained):

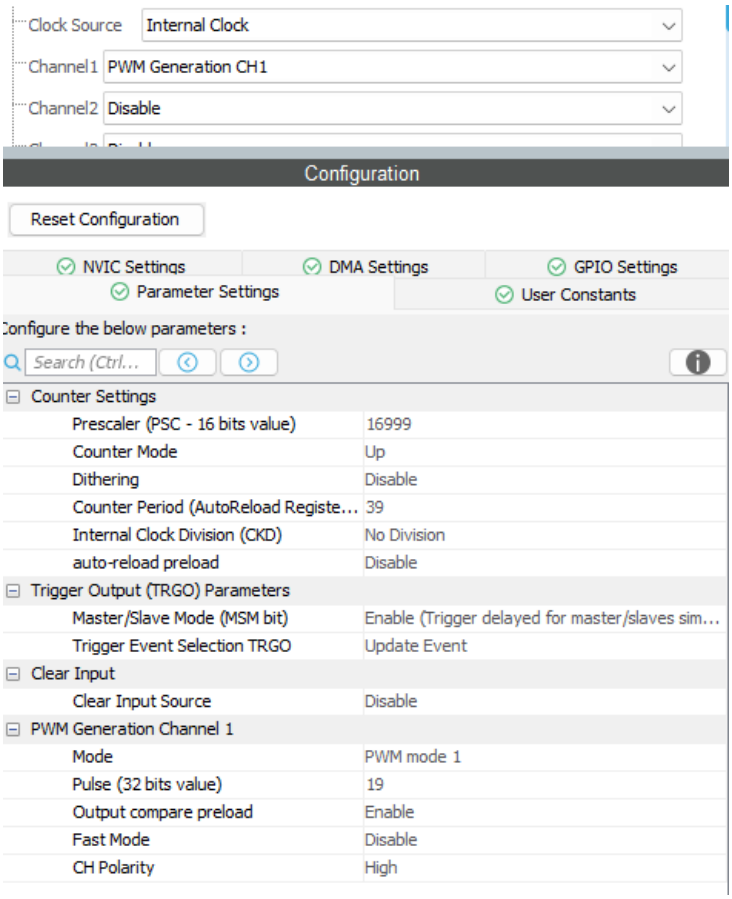



Figure 26: TIM2 Configuration

- **Clock Source**: Internal Clock
  - TIM2 uses the internal clock of the microcontroller.
  - **Advantage**: The internal clock source ensures stable and reliable operation without the need for external clock sources.
- **Channel 1**: PWM Generation CH1
  - TIM2 Channel 1 is configured for PWM signal generation.
  - **Advantage**: This enables TIM2 to generate precise PWM signals for controlling LEDs in our case, in order to ensure the blinking following HR as requested.

- **Prescaler (PSC)**: 16999
  - The prescaler value divides the clock frequency by 16999+1. This was done assuming that we wanted an high frequency of samples for each second (about 250). The formula used was the following:

$$UpdateEvent = \frac{Timer_{Clock}}{(Prescaler + 1)(Period + 1)} = \\ = \frac{170 \cdot 10^6}{(16999 + 1)(39 + 1)} = 250\ \text{Hz}$$

    The samples acquired by us are **250** ate each second, wich is enough for our application.
  - **Advantage**: Reduces the clock frequency in order to manage the correct number of samples acquisition for the ADC.
- **Counter Period (AutoReload Register)**: 39
  - The counter resets after reaching the value of 39.
  - **Advantage**: Defines the period , it was choosen 39 in order to reach 250Hz sampling freqeuncy.
- **Master/Slave Mode (MSM)**: Enable
  - Trigger delayed for master/slave synchronization.
  - **Advantage**: Allows synchronization with ADC, which can be crucial for coordinated data acquisition.

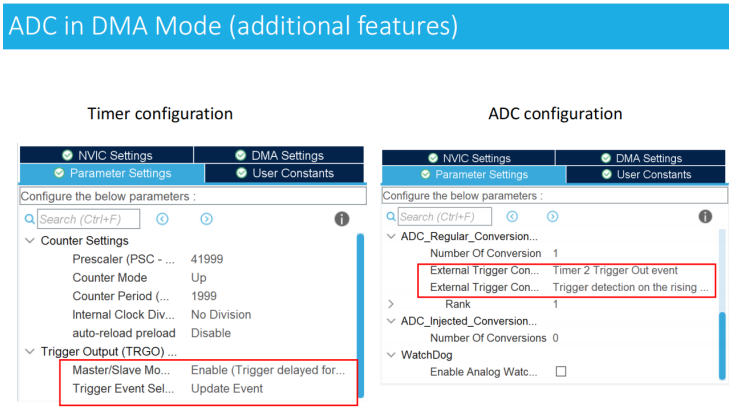



Figure 27: Example of usage Timer to use the ADC in DMA mode

- **Pulse (32 bits value)**: 19
  - The pulse width is set to 19.
  - **Advantage**: Defines the duty cycle of the PWM signal (In a periodic signal, the duty cycle is the percentage of one period), which can be adjusted to control the power delivered to the load.

### Timers Configuration Details: TIM3

The configuration of TIM3 was crucial for managing precise timing operations and synchronizing various tasks within the project. Below are the detailed settings used for TIM3:

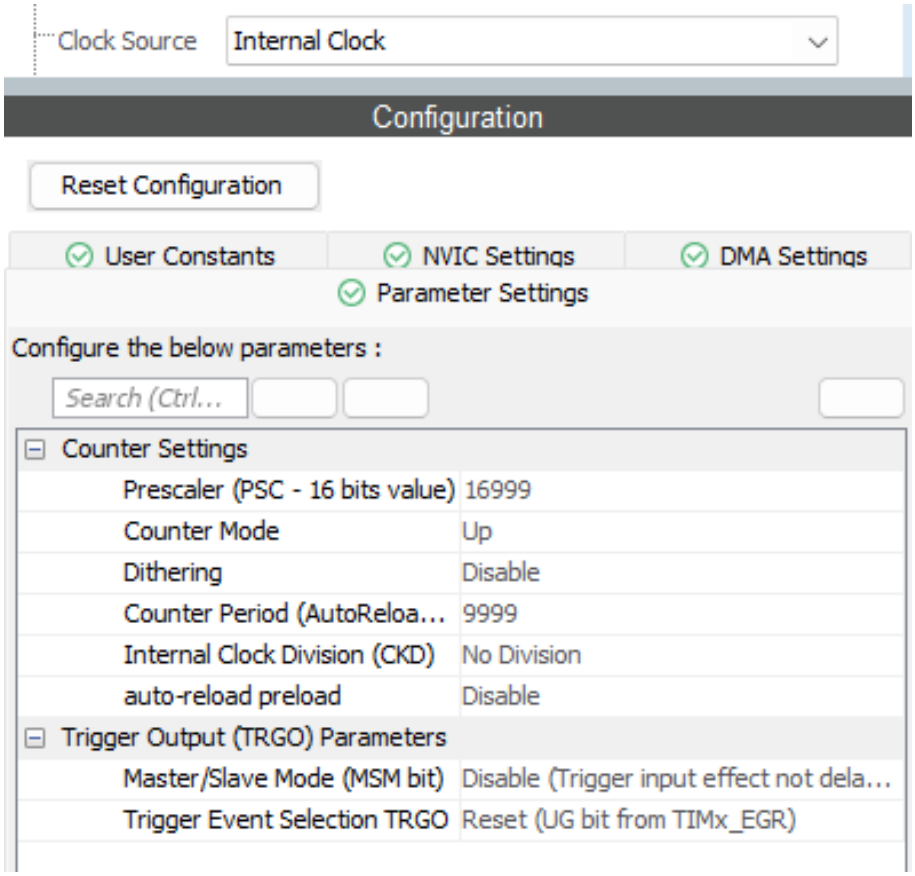



Figure 28: TIM3 Configuration

- **Clock Source**: Internal Clock
  - TIM3 uses the internal clock of the microcontroller.
  - **Advantage**: The internal clock source ensures stable and reliable operation without the need for external clock sources.
- **Prescaler (PSC)**: 16999
  - The prescaler value divides the clock frequency by 16999+1.
  - **Advantage**: Reduces the clock frequency to a suitable value for the desired timing operations, providing finer control over the timing intervals.

- **Counter Period (AutoReload Register)**: 9999
  - The counter resets after reaching the value of 9999.
  - **Advantage**: Defines the period of the timer, which can be adjusted to achieve the desired timing interval. It is important to denote that in this case, we needed to use a timer to perform certain actions after an established period (30 seconds to calculate **HR** and **HRV** in **Home Mode** and 5 seconds to update **HR** and **HRV** graphics in **Advanced Mode**). We decided to update the timer every second and to use a timer in the code. To achieve this, we used the following formula:

$$UpdateEvent = \frac{Timer_{Clock}}{(Prescaler + 1)(Period + 1)} = \\ = \frac{170 \cdot 10^6}{(16999 + 1)(9999 + 1)} = 1\,\text{Hz}$$

    This allows the timer to be updated every second. Moreover a timer operating at a lower frequency (1Hz) consumes less energy compared to one operating at higher frequencies. This is particularly beneficial in order to have some benefit in terms of power efficiency.
- **Master/Slave Mode (MSM)**: Disable
  - The trigger input is not delayed for master/slave synchronization.
  - **Advantage**: Allows TIM3 to function independently without being synchronized with other timers, providing greater flexibility.

### USART2 Configuration Details

The configuration of the USART (Universal Synchronous/Asynchronous Receiver/Transmitter) was essential for reliable and efficient serial communication in our project. It has been used to transmit data and display them on a serial plotter or in a terminal. Below are the detailed settings used for the USART:

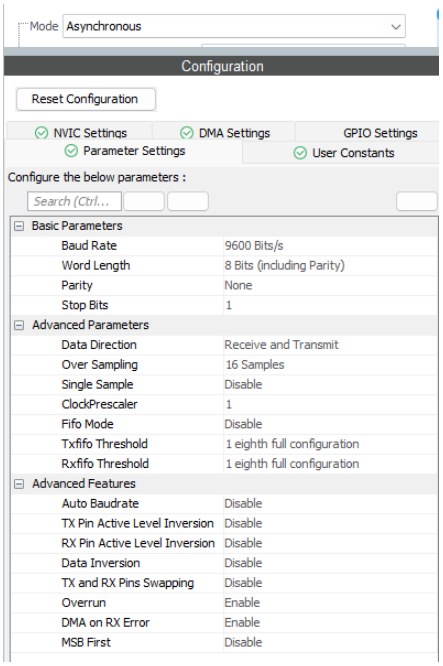



Figure 29: USART2 Configuration

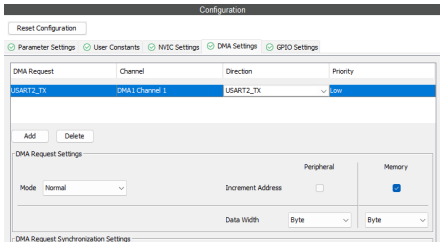



Figure 30: USART2 DMA Configuration

- **Asynchronous**
  - The USART operates in asynchronous mode.
  - **Advantage**: Asynchronous mode is suitable for standard serial communication where timing is not synchronized with a clock signal, providing flexibility in various communication scenarios.
- **Baud Rate**: 9600 Bits/s
  - The baud rate is set to 9600 bits per second.
  - **Advantage**: A standard baud rate that ensures compatibility with many devices and systems, providing a balance between speed and reliability. For our application, that involve the transmission of sensor data and control commands, 9600 bits per second provides sufficient speed. It has been choosen for the project to ensure timely data transmission without unnecessary bandwidth consumption.

- **Data Direction**: Receive and Transmit
  - The USART is configured for both receiving and transmitting data.
  - **Advantage**: Enables full-duplex communication, allowing for simultaneous data transmission and reception.
- **Clock Prescaler**: 1
  - The clock prescaler is set to 1.
  - **Advantage**: No division of the clock, maintaining the highest possible clock frequency for USART operations.
- **Word Length**: 8 Bits (including Parity)
  - Each data frame consists of 8 bits.
  - **Advantage**: Common word length for serial communication, ensuring compatibility and simplicity in data handling.
- **Parity**: None
  - No parity bit is used.
  - **Advantage**: Simplifies the data frame and eliminates the need for parity checking, reducing overhead.
- **Stop Bits**: 1
  - One stop bit is used.
  - **Advantage**: Standard configuration that provides a balance between ensuring proper frame termination and minimizing the transmission time.
- **Use of DMA**:
  - The UART configuration utilizes DMA for data transmission.
  - Reason for Using DMA in Transmission: In our project, DMA was used for USART transmission to efficiently handle large data blocks without burdening the CPU. This ensures that data is transmitted continuously and reliably, especially when sending large amounts of data to a serial plotter or terminal. Using DMA for transmission offloads the CPU from managing the data transfer, allowing it to focus on other critical tasks, which enhances overall system performance. Since reception typically involves smaller and less frequent data packets, it was managed directly by the CPU without the need for DMA.

### I2C3 Configuration Details

In our project, the use of I2C3 for both the SSD1306 OLED display and the DS3231 external clock ensures efficient and reliable data communication.

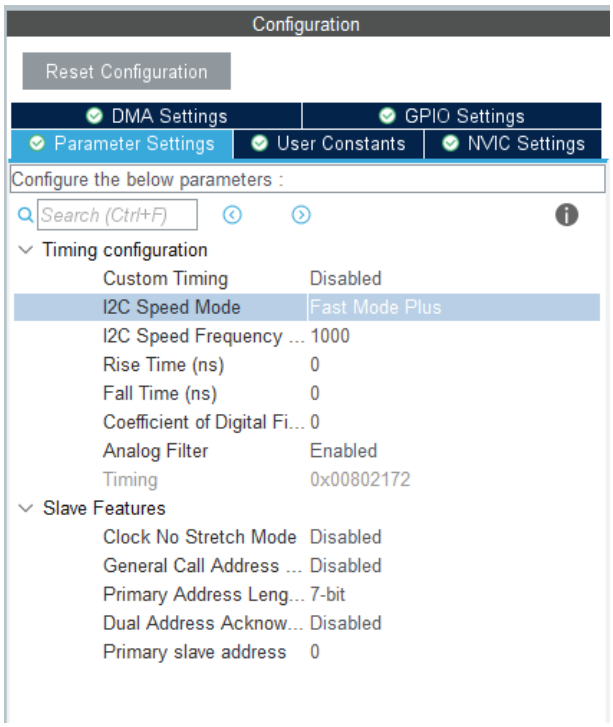



Figure 31: I2C3 Configuration

The following parameters have been configured and are interesting to know for the I2C3 interface:

- **I2C Speed Mode:** Fast Mode Plus
  - **Description:** Selects a high-speed mode for I2C, allowing speeds up to 1 MHz.
  - **Advantage:** Improves data transfer speed, reducing the time required for communications between devices.
- **I2C Speed Frequency:** 1000 kHz
  - **Description:** I2C3 communication frequency set to 1 MHz.
  - **Advantage:** This was a project choice and allowed a fast data transfer, which was very useful for plotting data acquired by ADC on the screen, without slowing down the sampling.

### 3.6.4 Fusion3D: Prototype Designing

To successfully create a prototype, it was essential to acquire knowledge of Fusion3D, a powerful CAD software used for 3D modeling and design. Fusion3D allows for the precise creation and modification of components, facilitating the development of complex prototypes with ease. This tool is particularly valuable in the design and prototyping phase of projects, providing an intuitive interface and a wide range of features for detailed modeling.

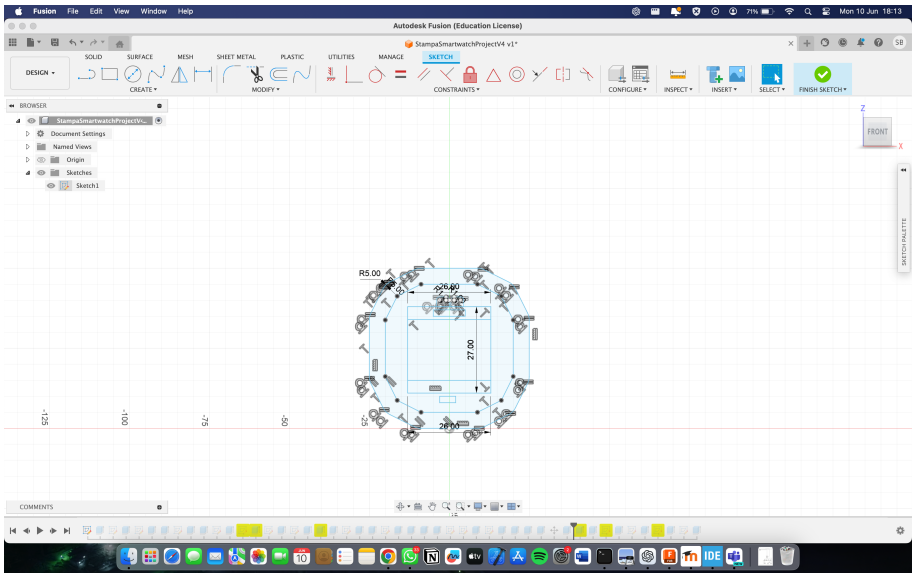



Figure 32: Fusion 3D: first Sketch

Using Fusion3D, we were able to visualize and refine our designs, ensuring that all components fit together perfectly before moving to physical production. This process not only saved time but also helped in identifying and resolving potential issues early in the development cycle.

(**NOTE**: The complete idea will be explained better in the next chapters)

# 4 Hardware Architecture

When tasked with bringing our project to life, our initial step was to assemble all components onto a breadboard, which we then simulated using Fritzing, a tool that simplifies the creation and testing of electronic circuits. This method allowed us to systematically visualize and refine our project concept in a structured manner.

## 4.1 Representing the Circuit

Fritzing is an open-source hardware initiative that makes electronics accessible as a creative material for anyone. It offers a robust and user-friendly platform for designing and documenting electronic projects. The software allows users to create circuit diagrams, design PCB layouts, and simulate the functionality of their projects in a virtual environment.

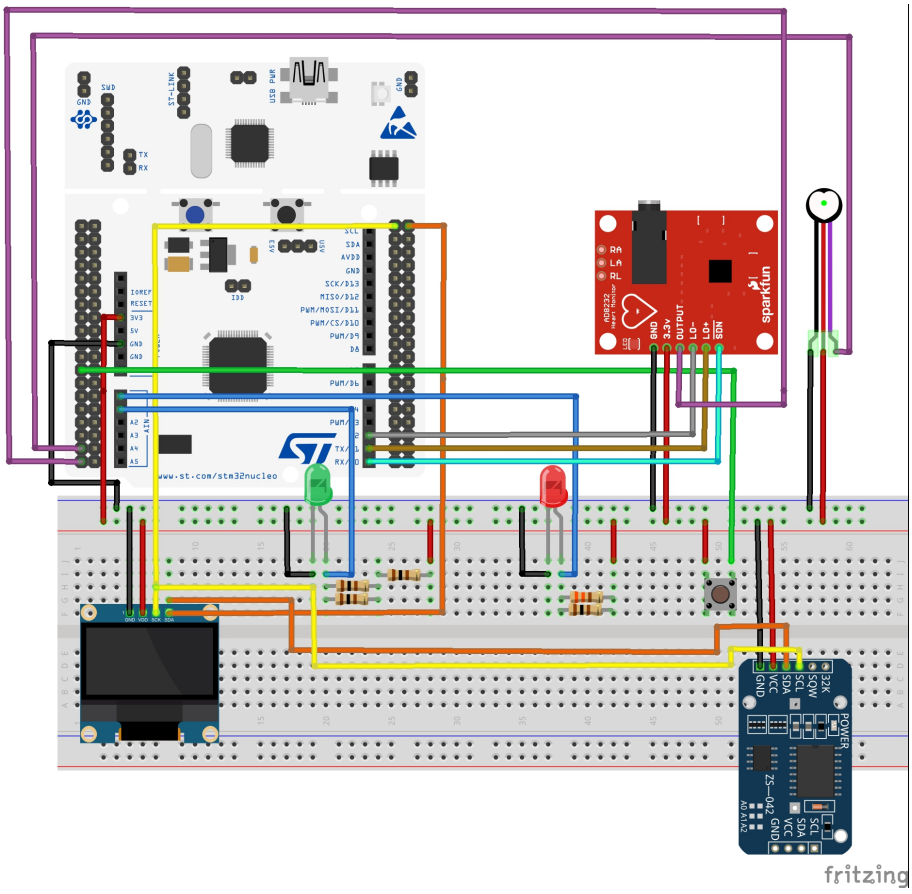



Figure 33: Fritzing Sketch

We began by selecting the necessary components from the library, including the NUCLEO microcontroller (a different version, but with the same configuration of pins to accurately represent our design), LEDs, and push button. Unfortunately, other components were not present in the standard library so we imported the following parts:

1. SSD1306 OLED display
2. DS3231 RTC module
3. AD8232 ECG sensor
4. PPG sensor

Using the drag-and-drop interface, we arranged these components on the virtual breadboard. The breadboard is powered by 3.3 V of the microcontroller and connected to the GND pin.

### 4.1.1 Projectual Hardware Choices

Once we placed the components on the virtual breadboard, we set up the I2C connections for the OLED display and RTC module. We used the SDA and SCL lines and connected them to the RTC module and OLED display directly. To make this work, we used the microcontroller's built-in pull-up resistors, which allowed both the RTC module and OLED display to connect to the same I2C bus without any issues.
In order to achieve the correct current flow and ensure safe operation we incorporated combinations of resistors in parallel and in series with the LEDs. The red LED operates at approximately **1.8V**, while the green LED at **2.1V**, and our board supplies **3.3V** As we can see from the following image.

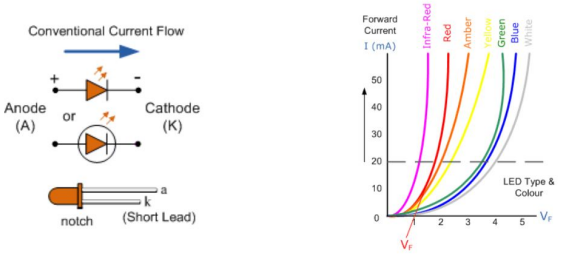



Figure 34: Led Type and Colour

To achieve this, we calculated the total resistance for each LED configuration. Using the formula:

$$R = \frac{V_{cc} - V_{LED}}{I}$$

where $\mathbf{V}_{cc}$ is the supply voltage, $\mathbf{V}_{LED}$ is the LED forward voltage, and **I** is the desired LED current, we computed the required resistor values for the red and green LEDs. For the RED LED, the precise values were:

$$R_{Red} = \frac{3.3V - 1.8V}{0.02A} = 75\ \Omega \qquad R_{Green} = \frac{3.3V - 2.1V}{0.02A} = 60\ \Omega$$

**RED LED:**

For the red LED, we used a parallel combination of two resistors: one with a value of 330 Ω and another with a value of 100 Ω.

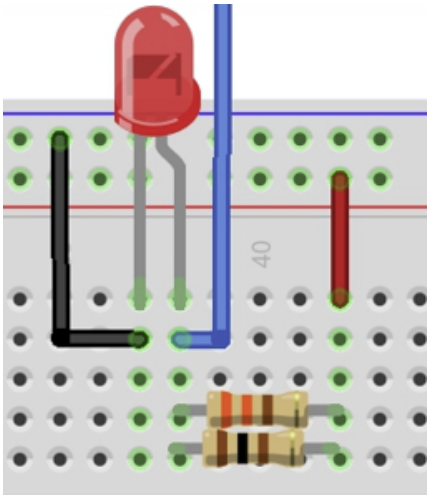


Figure 35: Circuit for Red Led

This resulted in an effective resistance of approximately 75 Ω, matching the calculated value. The equivalent resistance $R_{eq}$ of resistors in parallel is calculated using the formula:

$$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2}$$

Substituting the resistor values:

$$\frac{1}{R_{eq}} = \frac{1}{330\ \Omega} + \frac{1}{100\ \Omega}$$

Solving for $R_{eq}$:

$$R_{eq} = \frac{1}{\frac{1}{330} + \frac{1}{100}} \approx 75\,\Omega$$

**GREEN LED:**

For the green LED, we used two resistors in parallel, both with a value of 100 Ω, alongside another resistor of 10 Ω in series.

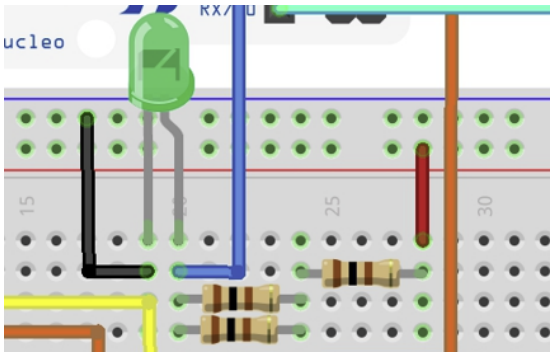



Figure 36: Circuit for Green Led

This configuration yielded an effective resistance of approximately 65 Ω, aligning with the calculated value. The equivalent resistance $R_{eq}$ of resistors in parallel followed by a resistor in series is calculated using the formulas:

$$\frac{1}{R_{parallel}} = \frac{1}{R_1} + \frac{1}{R_2}$$

Substituting the parallel resistor values:

$$\frac{1}{R_{parallel}} = \frac{1}{100\,\Omega} + \frac{1}{100\,\Omega}$$

Solving for $R_{parallel}$:

$$R_{parallel} = \frac{1}{\frac{1}{100} + \frac{1}{100}} = 50\,\Omega$$

Adding the series resistor:

$$R_{eq} = R_{parallel} + R_3 = 50\,\Omega + 10\,\Omega = 60\,\Omega$$

Therefore, the effective resistance is approximately 60 Ω, matching the calculated value.

**BUTTON**

We integrated a button into the circuit, adding the built-in pull-down resistor of the NUCLEO board. This resistor ensures that the input pin remains at a defined logic level (LOW) when the button is not pressed, thereby avoiding floating values.

**PPG and ECG sensors**

It is also intentionally mapped the AD8232 and PPG sensor outputs to the pins corresponding to ADC2 channels 9 and 8 respectively. This deliberate alignment enhances the efficiency of data acquisition, ensuring that analog signals from these sensors are interfaced with the same ADC.

Lastly, we connected the other pins of the AD8232 sensor to GPIO pins for efficient control and communication. Specifically, we've linked the LO-, LO+, and SDN pins of the AD8232 to GPIO pins PA10, PC4, and PC5 respectively. This deliberate mapping allows for precise monitoring and control of the sensor's functions.

However, as we delved deeper into the project, we recognized the opportunity to explore a more innovative and hands-on approach. We pivoted towards crafting a smartwatch prototype, a creative endeavor that allowed us to transcend virtual simulations and bring our ideas to life in a physical form.

## 4.2 The Idea: Smartwatch Prototype

By building a smartwatch prototype, we aimed to bridge the gap between concept and reality, immersing ourselves in the practicalities of hardware design and integration. This decision enabled us to explore the intricacies of wearable technology. In our smartwatch prototype, we meticulously arranged the components to optimize functionality and user experience. Positioned on the front face of the prototype are two LEDs and a vibrant screen, serving as primary indicators and display interfaces. These elements provide essential feedback and information to the user at a glance.

Meanwhile, on the back of the prototype resides the PPG sensor, strategically placed to facilitate accurate heart rate and blood oxygen level monitoring. This placement ensures optimal sensor performance while maintaining user comfort and convenience. Additionally, we integrated a push button on the side of the prototype, enabling intuitive user interaction and navigation through menus and functionalities. This ergonomic placement enhances usability and accessibility, allowing users to interact with the device effortlessly.

By thoughtfully arranging the components (LEDs and screen on the front, PPG sensor on the back, and push button on the side) we aimed to optimize functionality, usability, and user comfort in our smartwatch prototype. This meticulous design approach ensures a seamless and intuitive user experience while leveraging the full potential of the integrated components.

It is important to note that in order to achieve a balance between wearability and usability, significant engineering efforts were made to the prototype. This involved careful consideration of design elements, and ergonomic factors to ensure that the prototype not only functions effectively but is also comfortable and practical for users to wear and use; considering that the same circuit showed at 4.1 hadto be bulid inside a small space. The following images show how the conceptual design has evolved:

It was projected a first trial mockup in order to understand what to improve.

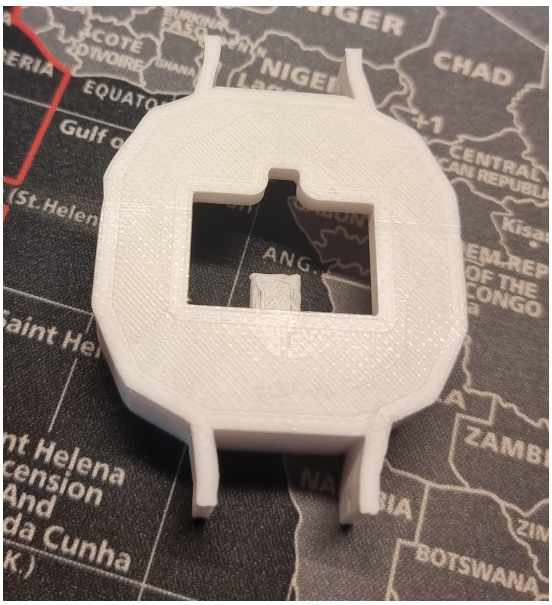



Figure 37: First Prototype

Figure 38: First Prototype

It was not perfect at all, as expected. However, this process helped identify the necessary improvements, leading to the following results:

Figure 39: Second Prototype

Figure 40: Second Prototype

The display, LEDs, the button, the PPG sensor, and the wristband fit perfectly, so the next steps were to create the entire circuit inside the mockup.

Figure 41: Second Prototype

Figure 42: Second Prototype

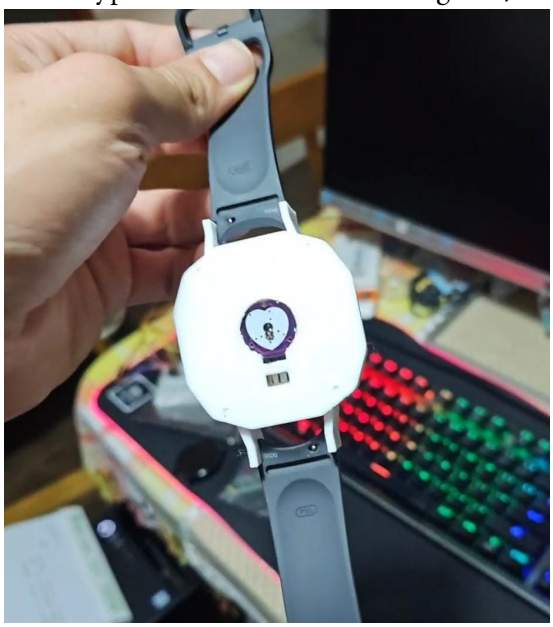


Figure 43: Second Prototype

This part was the most difficult because the entire circuit had to be redesigned to fit into a circular and restricted area. The challenge required meticulous planning and precision to ensure that all components were arranged efficiently without compromising functionality. Every element had to be carefully positioned to maintain connectivity and performance within the confined space.

Figure 44: Building The Circuit: Display

First of all, the display was tested and confirmed to be working perfectly. With the display functioning correctly, the next step was to design the rest of the circuit around it. The layout had to be carefully planned to accommodate the display at the center, ensuring that all other components were arranged in a circular and compact manner to fit within the limited space.

Figure 45: Building The Circuit: LEDs

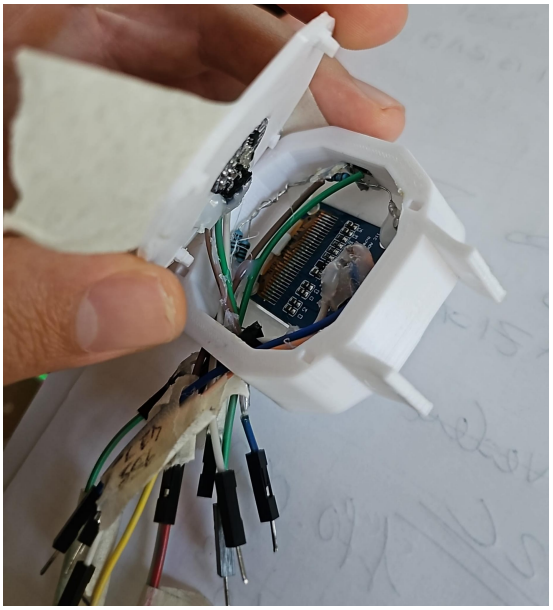


Figure 46: Full Circuit

The LEDs were placed strategically, ensuring that the necessary resistors were used.

After that, the push button and the PPG sensor were positioned appropriately. All cables were carefully routed through specially created leftside hole to maintain a neat and organized layout.

Figure 47: Testing part: Full Circuit

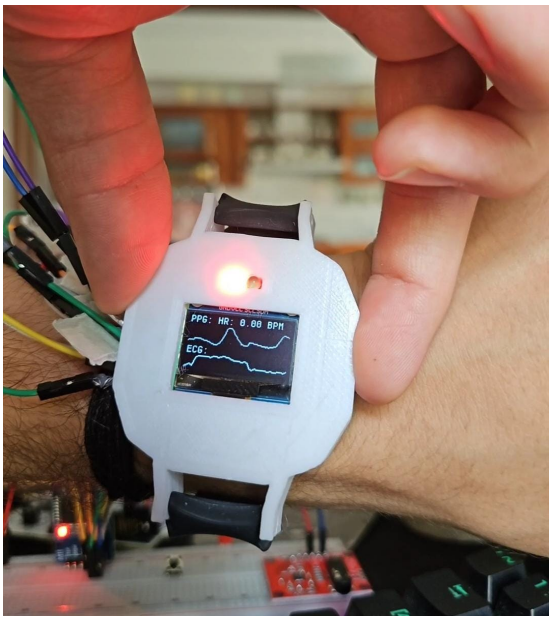



Figure 48: Testing part: Full Circuit

Finally, the entire circuit was tested thoroughly to ensure its functionality. After rigorous testing, we confirmed that the prototype works perfectly. DONE!!! IT REALLY WORKS.

**Note1:** Due to the restricted space, the **RTC** and **ECG** sensors were left outside the prototype.

**Note2:** It might not be evident from the sketches, but the entire circuit was made safe. All cables and unwrapped resistors were properly insulated.

5

# Software Architecture

For our Software Architecture, we've designed a structure that efficiently manages data from the PPG and ECG sensors. These structures are pivotal for organizing and processing sensor data effectively.
By adhering to consistent coding practices and modular design principles, we promote code clarity, ease of debugging, and easy collaboration among developers. Each source file encapsulates a specific set of functionalities, fostering code reuse and facilitating future modifications and enhancements.

Our codebase is organized into source files and their corresponding headers, facilitating modularity and maintainability. The source files are:

- **DataStructure.c**
- **ds3231.c**
- **graph.c**
- **ModeManager.c**
- **ssd1306.c**

## 5.1 DataStructure.h

The DataStructure.h header file encapsulates essential data structures and function prototypes necessary for managing **sensor data** and **heart metrics** within our software system.
Having all these components consolidated in a single header file offers several advantages. Firstly, it promotes modularity and organization in our codebase by centralizing the definitions of data structures and function prototypes related to sensor data and heart metrics management. This modular approach enhances code readability and maintainability, as developers can easily locate and reference these essential components when working on different parts of the software.
Moreover, by encapsulating data structures such as Signal and HeartMetrics, along with their associated functions, within a single header file, we establish a clear interface for interacting with sensor data and heart metrics throughout the software.
Additionally, consolidating all sensor data management functionalities in one header file promotes code reuse and reduces redundancy.

**DataStructure.h**

```
const MAX_VALUES
const SIGNAL_NAME_MAX_LENGTH
const PLOT_INTERVAL

Signal{
   values : float *
   filtered_values : float *
   measurement_time[MAX_VALUES]  : Date_Time
   num_values : int
   min_value : float
   max_value : float
   average_value : float
   peaks : int *
   num_peaks : int
   count_bad_sensor : int
   signal_name[SIGNAL_NAME_MAX_LENGTH] : char
   sampling : int
   window_sampling : int
   disconnected_flag : int
}

HeartMetrics {
   heart_rate : int *
   hrv : float *
   measurement_time : Date_Time *
   num_metric : int
}
```

```
init_HeartMetrics(HeartMetrics *metric) : void
init_signal(Signal *signal, float min_value, float max_value, const char *signal_name) : void
realloc_values(Signal *signal) : void
add_signal_values(Signal *signal, float value) : void
add_heart_metrics(HeartMetrics *metric,  Signal *s) : void
get_num_values(Signal *signal) :int
process_data(float *buffer, Graph *graph, uint8_t currentMode, Signal *signal) : void

calculate_mean(Signal *signal) : float
calculate_filtered_mean(Signal *signal) : float
calculate_variance(Signal *signal) : float
calculate_std_dev(fSignal *signal) : float

filter_ppg_signal(Signal *ppg) : void
filter_ecg_signal(Signal *ecg) : void
moving_average_filter(Signal *ppg,int window_size) : void
low_pass_filter(Signal *signal, float alpha) : void
high_pass_filter(Signal *signal, float alpha) : void


check_signal(Signal *signal) : int

detect_peaks(Signal *signal) : int
calculate_heart_rate(int *peaks, int num_peaks) : int
calculate_hrv(int *peaks, int num_peaks) : float

display_systolic_diastolic(Signal *ppg) : void
display_pulse_transit_time(Signal *ecg, Signal *ppg) : void
display_heart_rate(float heart_rate, int x, int y) : void
update_hr_hrv_graphs( Graph *hr_graph, Graph *hrv_graph, int new_hr, float new_hrv) : void
```

Figure 49: DataStucture.h

## 5.2 ds3231.h

The ds3231.h header file provides essential definitions and function prototypes for interfacing with the DS3231 real-time clock (RTC) module within our software system. This header file serves as a central hub for **managing communication** with the DS3231 RTC module, encapsulating functionalities such as reading the current date and time, setting up the RTC module, and converting between binary-coded decimal (BCD) and decimal formats for timekeeping data.
Moreover, the ds3231.h header file establishes a clear interface for interacting with the RTC module, abstracting away low-level details of communication protocols and data formats. This abstraction simplifies the integration of the module into our software system and promotes consistency in how RTC-related operations are performed throughout the application.

## 5.3 graph.h

For the graphical representations, our files facilitate the creation, modification, and visualization of the graphs using the SSD1306 display. By encapsulating graph-related functionalities in this files, we promote reusability, and we can easily integrate graph functionality into different modules of the application without duplicating code, as we did by using them for both PPG and ECG.

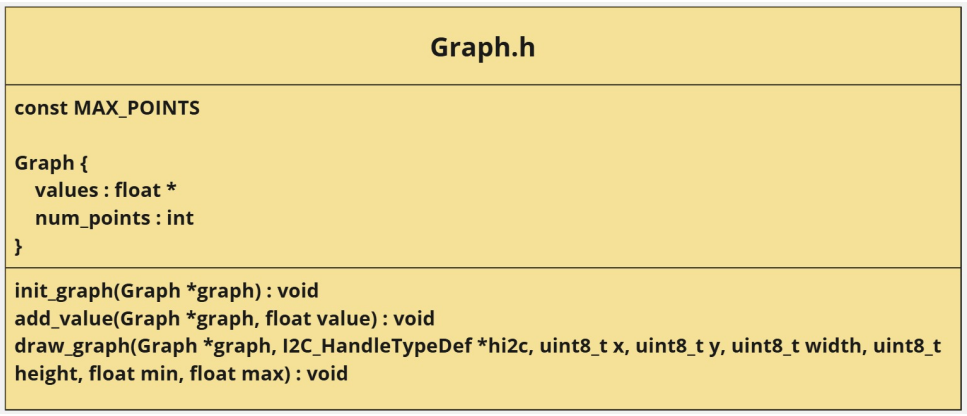



Figure 50: graph.h

## 5.4 ModeManager.h

The ModeManager files act as a centralized control mechanism for managing different operational modes within our software system. By encapsulating mode-related functionalities into this file, we achieve better code organization and enhances readability, as it can be easy to modify mode-specific logic without going in the main program. Separating mode management into its own header file promotes abstraction, allowing us to focus on high-level system behavior rather than low-level implementation details. Additionally, by defining mode-related constants and functions in the header file, we establish a clear and standardized interface for interacting with different modes throughout the software.

## 5.5 ssd1306.h

Integrating the SSD1306 library into our project allows us to interface with SSD1306 OLED displays on STM32 microcontrollers using HAL I2C libraries.
This library provides essential functionalities for initializing, controlling, and displaying content on the SSD1306 display. With its customizable options and flexible interface, it allowed us to design and implement the graph files tailored to our project's specific requirements. By utilizing functions such as drawing pixels, lines, rectangles, and bitmaps, along with supporting multiple font sizes for text display, we can create dynamic graphical representations directly on the OLED screen, like adding the icon we created to make the Home Mode more visually appealing.
It not only simplifies the process of interfacing with OLED displays but also **expands** the scope of our project by enabling the creation of custom graphs and visualizations. This addition enriches the user experience and enhances the overall functionality of our application, transforming the OLED display into a versatile and informative output medium.

# 6 Software Interfaces

During the development of the project, the software interfaces play a crucial role in bridging the gap between hardware components and application-level software. In our case, fundamental was the **Hardware Abstraction Layer (HAL)**. The HAL provides a standardized API for interacting with the underlying hardware, abstracting the complexities of direct hardware manipulation and enabling developers to write more portable and maintainable code.

The HAL, provided by STM32, serves as an intermediary between the hardware and the higher-level software, offering a collection of drivers and libraries that simplify hardware access. This abstraction is particularly valuable in reducing the dependency of application code on specific hardware implementations, thus enhancing code reusability and portability.

The primary functions of the HAL include:

- **Simplification of Hardware Access:** By providing a high-level API, the HAL reduces the need for detailed knowledge of hardware registers and low-level operations.
- **Portability**: Applications written using the HAL can be more easily ported to different hardware platforms, as the HAL abstracts away the hardware-specific details.
- **Code Maintenance:** The standardized interfaces provided by the HAL make it easier to update and maintain code, as changes in hardware do not necessarily require changes in the application code.
- **Efficiency:** The HAL provide optimized performnces.

The following paragraphs aim to explain how the HAL has been used and the choices made during its implementation. Some parts have already been perfectly described in Chapter 3.6.3. Therefore, there will be some repetitions, but not explained at the same level of detail.

## 6.1 ADC: acqusition of signals from PPG and ECG

We have already extensively explained the usage and functionality of the ADC. To integrate it into the project, it was configured using the HAL interface. The **Resolution** was set to 12 bits, ensuring high precision in the digital representation of analog signals. The **DMA Continuous Request** was enabled to facilitate uninterrupted data transfer without burdening the CPU. Additionally, the **Number of Conversions** was set to 2, allowing the ADC to handle multiple input channels efficiently. The **Cycles** were set to 6.5 to ensure rapid data acquisition, enhancing the responsiveness of the system.

Two channels were configured: one for the ECG sensor and one for the PPG sensor. This setup allows simultaneous monitoring of both physiological signals, ensuring comprehensive data collection and analysis.

To perform ADC operations in the code, it was first initialized using the function MX_ADC2_Init(). The ADC was then used in combination with timers in order to synchronize data acquisition. Specifically, when TIM2 detects a trigger event, the ADC starts acquiring data using the function:
`HAL_ADC_Start_DMA(&hadc2, (uint32_t*) data, ADC_NUM_CHANNEL).`

Using **DMA** for starting the ADC offers significant advantages. It allows the ADC to transfer data directly to memory without requiring CPU intervention for each sample. This eliminates the need to repeatedly call a function like HAL_ADC_GetValue() to retrieve the conversion result. By offloading data transfer to the DMA controller, the CPU is free to perform other tasks, improving overall system efficiency and performance. Additionally, this approach reduces the risk of data loss or corruption, as the DMA ensures timely and reliable data transfer.

## 6.2 TIM2: PWM and regulation of ADC Acquisition through DMA

The Timer TIM2 was used primarily to regulate the correct acquisition of data via DMA. The **period** was set to **39** and the **prescaler** was set to **16999** to achieve a sampling frequency of **250Hz**, using the formula 3.6.3

This configuration ensures that the ADC operates at a precise sampling frequency, allowing for accurate and consistent data acquisition. By using TIM2 to trigger the ADC, we synchronize data acquisition with the timer's intervals, ensuring that the data is sampled at regular, predictable intervals.

Moreover, TIM2 was essential for generating **Pulse Width Modulation** signals to control our LEDs. The PWM signals were synchronized with the signals acquired by the ADC, enabling dynamic control of the LED intensity based on the real-time data. This synchronization ensures that the LED's behavior accurately reflects the physiological signals being monitored

To perform TIM2 operations in the code it was initialized firstly thanks to the macro `MX_TIM2_Init()` .
Thanks to the functions `HAL_TIM_PWM_Start` and `HAL_TIM_PWM_Stop()` the PWM signal was started and stopped , respectively. Additionally, the following HAL macros were used for precise control over the PWM signal:

- `_HAL_TIM_GET_AUTORELOAD(&htim2) + 1) / 100.0f)`: This macro retrieves the auto-reload value of TIM2, which defines the PWM period. By dividing this value by 100.0f, a normalized PWM duty cycle percentage is obtained.
- _HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, pwm_value): This macro sets the compare value for TIM2 on channel 1, effectively adjusting the duty cycle of the PWM signal. pwm_value determines the brightness of the LED, allowing for fine-grained control based on the acquired data.

In particular, the function set_PWM_LED(Signal *signal), created by us, aims to handle the PWM by utilizing the previously mentioned macros. This function calculates the appropriate PWM value based on the input signal and adjusts the LED brightness

accordingly, ensuring real-time responsiveness and accurate representation of the physiological data being monitored.

The Timer TIM2 has been also recalled in the **callback function** of TIM3 but just to handle the **ADC acquisition**.
In that case the macro used was `HAL_TIM_Base_Start(&htim2)`.

## 6.3 TIM3: Generation of interrupt every second

The Timer TIM3 was used primarily to manage precise timing operations and synchronize various tasks within the project. The **period** was set to **9999** and the **prescaler** was set to **16999** to achieve a sampling frequency of **1Hz**, using the formula 3.6.3.

This configuration ensures that the timer generates an interrupt every second. By generating an interrupt at this interval, we can utilize a counter in the code to keep track of the passing seconds. This precise timing mechanism is crucial for various time-dependent operations within the project, such as updating displays, logging data, and triggering events at regular intervals.

To correctly use the timer TIM3 in interrupt mode, the HAL was used to enable interrupts and handle the callback associated with each timed event.
The function `MX_TIM3_Init()` was used to initialize the timer.
`HAL_TIM_Base_Start(&htim3)` was used to start it. Subsequently, to work with interrupts, the implementation of the callback function was necessary. This involved defining the `HAL_TIM_PeriodElapsedCallback()` function, which is called by the HAL library whenever the timer period elapses. This callback function allowed us to execute specific code every time the timer generated an interrupt, facilitating precise and periodic execution of tasks.

## 6.4 USART2: Debugging and plotting of datas

The configuration of the USART (Universal Synchronous/Asynchronous Receiver/Transmitter) was essential for reliable and efficient serial communication in our project, it allows the serial communication between NUCLEO-G474RE and the serial terminal. It has been used to transmit data and display them on a serial plotter or in a terminal.

To use the USART2, it was set in asynchronous mode with a baud rate of 9600 bits/s. The **word length** was configured to 8 bits, ensuring compatibility with standard serial communication protocols, parity to none, stop bit to 1. This configuration allows for reliable and efficient data transmission and reception between the microcontroller and external devices.

In order to use the USART the HAL was used to recall it using the function:
`HAL_UART_Transmit_DMA()`

## 6.5 I2C: SSD1306 and RTC

The I2C (Inter-Integrated Circuit) interface facilitates communication with external devices such as sensors, displays, and other components that utilize the I2C protocol. To control the OLED display, and the external clock, we configured the I2C interface using the HAL library. This involved setting the device addresses, transmission speed, and other configuration parameters necessary for proper communication with each device.

After the initial configuration, we employed HAL interface functions to transmit and receive data effectively.

7

# Coding Protocols

This chapter outlines the various design choices made during the development of the project. Each section details specific aspects of the project, including the initialization process, data measurement and processing, heart rate and variability calculations, and operational modes. These choices ensure the effective acquisition, processing, and display of physiological data, providing a comprehensive understanding of the system's functionality and implementation.

## 7.1 Initialization

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_ADC2_Init();
MX_TIM2_Init();
MX_I2C3_Init();
MX_USART2_UART_Init();
MX_TIM3_Init();
/* USER CODE BEGIN 2 */
ssd1306_Init(&hi2c3);

HAL_TIM_Base_Start_IT(&htim3);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);

initMode();

init_signal(&ppg, 0, 4095, "PPG");
init_signal(&ecg, 0, 4095, "ECG");

init_HeartMetrics(&heart_rate);

init_graph(&ppg_graph);
init_graph(&ecg_graph);
init_graph(&hr_graph);
init_graph(&hrv_graph);

current_ppg_buffer = (float*) malloc(PLOT_INTERVAL * sizeof(float));
processing_ppg_buffer = (float*) malloc(PLOT_INTERVAL * sizeof(float));
current_ecg_buffer = (float*) malloc(PLOT_INTERVAL * sizeof(float));
processing_ecg_buffer = (float*) malloc(PLOT_INTERVAL * sizeof(float));
```

Figure 51: Initialization

The presented code initializes and configures a series of peripherals on a microcontroller.

**Peripheral Initialization:** Various hardware modules such as GPIO, DMA, ADC, timers, I2C, and UART are initialized. These modules enable the microcontroller to interact with sensors, actuators, and other hardware components.

**SSD1306 Display Initialization:** The OLED SSD1306 display is initialized via the I2C bus. This display is used to show graphical information.

**Timer Configuration:**

- Timer 3 is started in base mode with interrupts, allowing the microcontroller to perform periodic actions.
- Timer 2 is started in PWM mode on channel 1, used to control the red LED. The PWM duty cycle is initially set to 0

**Operating Mode Initialization:** A function is called to configure the application's operating mode. As default, it starts with the Home Mode.

**PPG and ECG Signal Initialization:** We employ a specialized structure to manage the data from the PPG and ECG sensors. This structure is meticulously designed to handle various attributes and metrics associated with the physiological signals. The PPG and ECG structures are initialized with a range of values from 0 to 4095, corresponding to the ADC resolution of our microcontroller. It includes fields for raw data, filtered data, timestamps, statistical values, and peak detection. By employing this structure, we ensure efficient data organization, precise control over sampling rates, facilitation of time-series analysis, and robust handling of sensor connections. This systematic approach optimizes the overall performance and reliability of our system by providing a comprehensive framework for managing sensor data.

**Heart Metrics Initialization:** Metrics related to heart rate are initialized. These metrics include the heart rate itself and heart rate variability.

**Graph Initialization:** Several graphs are initialized to display PPG, ECG, heart rate, and heart rate variability data.

**Memory Allocation for Data Buffers:** Dynamic memory is allocated for the PPG and ECG data buffers. These buffers temporarily store data collected from the sensors for subsequent processing and display.

Double Buffering: As a design choice, we used double buffering for data acquisition, which allows the separation of acquisition from analysis and drawing processes. Specifically, we implemented four buffers: two for PPG and two for ECG. This ensures that acquisition is not interrupted.

**Minimizing Callback Time:** The ADC callback primarily handles sample collection and determines when to plot, minimizing the time spent on graph drawing operations. With these optimizations, the system is more efficient and capable of managing data acquisition and graph drawing without significant slowdowns.

The code prepares the microcontroller to acquire, process, and display physiological data by configuring various necessary hardware and software modules for the application's operation.

## 7.2 Measurement

### 7.2.1 TIM

```
Get_Date(&first_measure, &hi2c3);
HAL_TIM_Base_Start(&htim2);
HAL_ADC_Start_DMA(&hadc2, (uint32_t*) data_out,
ADC_NUM_CHANNEL);

ssd1306_DrawBitmap(42, 42, image, 16, 16, White);
ssd1306_UpdateScreen(&hi2c3);
```

Figure 52: TIM3 after 30 seconds

Once the program starts, the TIM3 begins to generate interrupts every second. Using a variable incremented each time it enters its callback, we are able to define a temporized behavior for which, after 30 seconds, the code configures the microcontroller to start collecting data from an ADC using DMA, starts a timer to synchronize operations, and displays an icon (a beating heart) on the SSD1306 OLED display. These operations prepare the system to acquire and display real-time data.

### 7.2.2 ADC

```
ppg.count_bad_sensor = 0;
ppg.disconnected_flag = 0;
add_signal_values(&ppg, data_out[0]);
filter_ppg_signal(&ppg);
if (check_signal(&ppg) == 1) {
    if (current_mode == 0)
        set_PWM_LED(&ppg);

    current_ppg_buffer[ppg_index++] =
            ppg.filtered_values[ppg.num_values];
    ppg_sample_count++;
    if (ppg_index >= PLOT_INTERVAL) {
        ppg_index = 0;
        ppg_buffer_full = 1;

        float *temp = current_ppg_buffer;
        current_ppg_buffer = processing_ppg_buffer;
        processing_ppg_buffer = temp;

        process_data(processing_ppg_buffer, &ppg_graph, current_mode,
                &ppg);
    }
    ppg.num_values++;
} else {
    ppg.count_bad_sensor++;
    if (ppg.count_bad_sensor > 25 && ppg.disconnected_flag == 0) {
        //Sensore scollegato
        ppg.disconnected_flag = 1;
        HAL_GPIO_WritePin(Led_Sensor_GPIO_Port, Led_Sensor_Pin,
                GPIO_PIN_SET);
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
    }
}

process_flag = 1;
```

Figure 53: ADC PPG

The ADC Callback function is crucial for processing the signals. When an ADC conversion is complete, the function checks if the PPG signal needs to be processed. If so, it reads the latest ADC value, which represents the PPG signal, and stores this value in the dedicated structure.

Next, it increments a counter that tracks the number of samples collected. The function then applies several filters to clean up the signal, removing noise and other artifacts. After filtering, it verifies whether the signal is within a valid range. If valid, the filtered signal is stored for further processing and analysis.

The function periodically updates the display with the latest PPG signal data, providing real-time feedback to users. If the buffer holding the PPG samples becomes full, the function reallocates memory to accommodate more samples, ensuring continuous data collection.

### 7.2.3 Filters

```c
void high_pass_filter(Signal *signal, float alpha) {
    signal->filtered_values[0] = signal->values[0];
    if (signal->num_values > 3) {
        signal->filtered_values[signal->num_values] = alpha
                * (signal->filtered_values[signal->num_values - 1]
                        + signal->values[signal->num_values]
                        - signal->values[signal->num_values - 1]);
    }
}

void low_pass_filter(Signal *signal, float alpha) {
    signal->filtered_values[0] = signal->values[0];
    if (signal->num_values > 3) {
        signal->filtered_values[signal->num_values] = alpha
                * signal->values[signal->num_values]
                + (1 - alpha) * signal->filtered_values[signal->num_values - 1];
    }
}

void moving_average_filter(Signal *ppg, int window_size) {
    float sum = 0.0;
    if (ppg->num_values < window_size) {
        ppg->filtered_values[ppg->num_values] = ppg->values[ppg->num_values];
        return;
    }
    for (int i = ppg->num_values - window_size; i <= ppg->num_values; i++) {
        sum += ppg->values[i];
    }
    ppg->filtered_values[ppg->num_values] = (sum) / window_size;
}

void filter_ppg_signal(Signal *ppg) {
    if (ppg->num_values > 0) {
        moving_average_filter(ppg, 5);
        high_pass_filter(ppg, 0.2);
        low_pass_filter(ppg, 0.95);
    } else {
        ppg->filtered_values[ppg->num_values] = ppg->values[ppg->num_values];

    }

}
```

Figure 54: Filters: High-Pass, Low-Pass and Moving Average

The filtering process for the PPG signal involves three main steps to clean up the raw data and ensure accurate analysis. First, a moving average filter smooths the signal by averaging a window of recent samples, which reduces short-term fluctuations and noise. Next, a high-pass filter removes low-frequency components such as baseline drift, retaining the relevant high-frequency components of the signal. Finally, a low-pass filter eliminates high-frequency noise, ensuring a cleaner signal for analysis. This was necessary in order to reach a signal as clean as possible.

When a new PPG sample is collected, the program first applies the moving average filter to smooth the signal by averaging the current and previous samples. Then, it uses the high-pass filter to remove low-frequency noise. Finally, the low-pass filter is applied to remove any remaining high-frequency noise. These filtering steps are implemented in the filter ppg signal function, which ensures that the PPG signal is clean and suitable for further processing and display.

Determining the optimal parameters for the filters was achieved through a process of experimentation and trial. By testing different parameter values and observing their effects on the signal, suitable settings were identified that effectively smoothed the signal while preserving its relevant features. This iterative approach allowed for fine-tuning the filter parameters to ensure optimal performance in cleaning up the PPG signal for accurate analysis.

## 7.3 Heart Rate and Heart Variability

### 7.3.1 TIM: PeriodElapsedCallback

```
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
HAL_TIM_Base_Stop(&htim2);
HAL_ADC_Stop_DMA(&hadc2);
timer3_counter = 0;

ssd1306_DrawBitmap(42, 42, image, 16, 16, Black);
ssd1306_UpdateScreen(&hi2c3);

heart_rate.heart_rate[heart_rate.num_metric] = detect_peaks(
        &ppg);
heart_rate.hrv[heart_rate.num_metric] = calculate_hrv(ppg.peaks,
        ppg.num_peaks);
heart_rate.num_metric++;
sprintf(msg, "peaks: %d \t window: %d \n\r", ppg.num_peaks,
        ppg.window_sampling);
HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), 100);
```

Figure 55: TIM3 after sampling time

Once 10 seconds (our sampling time) are up, the code performs several critical tasks for managing real-time data acquisition and display on a microcontroller. Initially, it halts the PWM signal generation and stops the associated timer. Simultaneously, it ceases the ADC data acquisition process facilitated by DMA. The OLED display is updated to clear a previously displayed bitmap, indicating a transition in visual content. Additionally, the code processes physiological data related to heart rate monitoring, capturing peaks in the PPG signal and calculating heart rate variability (HRV). Finally, it transmits diagnostic information via UART, providing insights into peak counts and sampling window parameters.

### 7.3.2 Heart Rate

The computations for heart rate and HRV are done by the detect peaks and calculate hrv functions.

The detect peaks function identifies peaks in the filtered PPG signal. It does this by iterating through the signal data, looking for local maxima that are higher than their neighboring points and exceed a certain threshold based on the average signal value. These peaks are stored in the peaks array of the Signal struct. The function then calculates the heart rate by counting the number of peaks within the specified window and converting it to beats per minute (BPM).

```
int detect_peaks(Signal *signal) {
    free(signal->peaks);
    signal->peaks = (int*) malloc(MAX_VALUES / 5 * sizeof(int));
    if (signal->num_values <= 2)
        return 0;

    int count = 0;


    for (int i = 1; i < signal->num_values - 1; i++) {
        if (signal->filtered_values[i] > signal->filtered_values[i - 1]
                && signal->filtered_values[i] > signal->filtered_values[i + 1]
                && signal->filtered_values[i] > 1.25 * signal->average_value) {
            if (count == 0 || (i - 1 != signal->peaks[count - 1])) {
                signal->peaks[count] = i;
                count++;
                i += 20;
            }
        }

    }
    signal->sampling = 0;
    if (count > 0)
        signal->num_peaks = count;
    return signal->num_peaks * 60 / signal->window_sampling;
}
```

Figure 56: Detecting Peaks

### 7.3.3 Heart Rate Variability

The calculate hrv function determines heart rate variability based on the intervals between consecutive peaks. It calculates the mean interval duration and then computes the variance of these intervals. The square root of the variance provides the HRV measure, indicating the variation in the time intervals between heartbeats. If there are fewer than five peaks, the HRV is set to a default value of 0.1 for stability.

```
float calculate_hrv(int *peaks, int num_peaks) {
    if (num_peaks <= 5)
        return 0.1;

    float intervals[num_peaks - 1];
    for (int i = 1; i < num_peaks; i++) {
        intervals[i - 1] = peaks[i] - peaks[i - 1];
    }

    float sum = 0;
    for (int i = 0; i < num_peaks - 1; i++) {
        sum += intervals[i];
    }
    float mean = sum / (num_peaks - 1);

    float variance = 0;
    for (int i = 0; i < num_peaks - 1; i++) {
        variance += (intervals[i] - mean) * (intervals[i] - mean);
    }
    variance /= (num_peaks - 1);

    float hrv = sqrt(variance);

    return hrv;
}
```

Figure 57: Computing heart rate variability

## 7.4 Waveform Mode

In waveform mode, the system samples PPG and ECG data, incrementing their respective sampling counters. If either counter exceeds 10, the system resets the timer and retrieves the current date and time from the latest PPG measurement. It then calls "add_heart_metrics()" to update the HR and HRV based on the ECG data. When "add_heart_metrics()" is called, it gets the current date and time, detects peaks in the ECG signal to determine heart rate, and calculates HRV from the intervals between these peaks. The new heart rate and HRV values are stored, and a message with the number of detected peaks and sampling window size is sent via UART.

### 7.4.1 ADC_ConvCpltCallback: Not Home Mode

```
else {
  add_signal_values(&ecg, data_out[1]);

  if (current_mode != MODE_HOME) {
      set_PWM_LED(&ecg);
      filter_ecg_signal(&ecg);
      if (HAL_GPIO_ReadPin(LO_Plus_GPIO_Port, LO_Plus_Pin)
              == GPIO_PIN_RESET
              && (HAL_GPIO_ReadPin(LO_Minus_GPIO_Port, LO_Minus_Pin)
                      == GPIO_PIN_RESET)) {

          current_ecg_buffer[ecg_index++] =
                  ecg.filtered_values[ecg.num_values];
          ecg_sample_count++;
          if (ecg_index >= PLOT_INTERVAL) {
              ecg_index = 0;
              ecg_buffer_full = 1;

              float *temp = current_ecg_buffer;
              current_ecg_buffer = processing_ecg_buffer;
              processing_ecg_buffer = temp;

              process_data(processing_ecg_buffer, &ecg_graph,
                      current_mode, &ecg);
          }
          ecg.num_values++;
          ecg.count_bad_sensor = 0;
          ecg.disconnected_flag = 0;
      } else {
          ecg.count_bad_sensor++;
          if (ecg.count_bad_sensor > 15 && ecg.disconnected_flag == 0) {
              ecg.disconnected_flag = 1;
          }
      }

      Date_Time d = ppg.measurement_time[ppg.num_values - 1];
```

Figure 58: ADC ECG

The code manages the reception and processing of ECG signal data. Initially, it adds the ECG signal values to the 'ecg' object using incoming data. It checks if the current mode is not 'HOME', then adjusts the LED PWM based on the ECG using `set_PWM_LED()`. It filters the ECG signal to enhance its quality using `filter_ecg_signal()`. It verifies the validity of the ECG signal through `check_signal()` and checks if GPIO

pins are at a low level. If the signal is valid, it stores the filtered ECG value in current_ecg_buffer, increments 'ecg sample count', and if necessary, sets 'ecg buffer full'. It manages any invalid or disconnected signals by updating 'ecg.count bad sensor' and 'ecg.disconnected flag'.

### 7.4.2 Graphs

```
void process_data(float *buffer, Graph *graph, uint8_t currentMode,
        Signal *signal) {
    // Esegui il processamento del buffer
    for (int i = 0; i < PLOT_INTERVAL; i++) {
        add_value(graph, buffer[i]);

    }

    if (currentMode == MODE_WAVEFORM) {
        if (strcmp(signal->signal_name, "PPG") == 0) {
            draw_graph(graph, &hi2c3, 0, 10, 128, 26, signal->max_value,
                    signal->min_value); // Grafico PPG in alto
        } else if (strcmp(signal->signal_name, "ECG") == 0) {
            draw_graph(graph, &hi2c3, 0, 45, 128, 18, signal->max_value,
                    signal->min_value); // Grafico ECG in basso
        }

        ssd1306_UpdateScreen(&hi2c3);
    }
}
```

Figure 59: Process Data

The `process_data()` function is pivotal within the physiological monitoring system, handling the real-time processing and display of PPG and ECG signal data. It is instrumental in providing a graphical representation of the patient's cardiac activities.

During operation, this function iterates through a buffer of floating-point data, incorporating each value into a specified graph structure. This process is conducted within a loop defined by a set interval, ensuring continuous updates to the graph using an appropriate function for adding values.

Graphical output is managed based on the current operational mode of the application. Specifically, when operating in 'waveform mode,' the function is designed to render the graph directly onto the OLED screen. The parameters defining the dimensions and positioning of this graph are dynamically set to optimize visual clarity and relevance.

## 7.5 Advanced Mode

```
} else if (current_mode == MODE_ADVANCED) {
    ecg.window_sampling++;
    ppg.window_sampling++;
    if ((timer3_counter > 5 || ppg.window_sampling > 5) &&
            (ecg.disconnected_flag == 1 || ecg.window_sampling > 5)) {
        if (ecg.disconnected_flag == 0) {
            add_heart_metrics(&heart_rate, &ecg);
        } else {
            add_heart_metrics(&heart_rate, &ppg);
        }
        update_hr_hrv_graphs(&hr_graph, &hrv_graph,
                heart_rate.heart_rate[heart_rate.num_metric - 1],
                heart_rate.hrv[heart_rate.num_metric - 1]);

        display_systolic_diastolic(&ppg);

        display_pulse_transit_time(&ecg, &ppg);

        timer3_counter = 0;
        ppg.sampling = 0;
        ppg.window_sampling = 0;
        ecg.sampling = 0;
        ecg.window_sampling = 0;
        if (ppg.num_values != 0)
            realloc_values(&ppg);
        if (ecg.num_values != 0)
            realloc_values(&ecg);
    }
}
```

Figure 60: Advanced Mode

In the "Advanced Mode" of our application, we focus on detailed processing and plotting of heart rate and heart rate variability. Heart rate is primarily acquired using the ECG sensor. If the ECG sensor is disconnected, the system switches to using the PPG sensor. This dual-sensor approach ensures reliable heart rate monitoring.
The system regularly checks sampling counters for ECG and PPG signals. If certain conditions are met, the heart rate data is processed. HR and HRV graphs are updated with the latest metrics. It is important to note that for projectual reasons the **HRV** graph is **commented out** and not plotted due to the few space on the OLED, it is prepared for **future use**.

The system analyzes the PPG signal to calculate and display the durations of systolic and diastolic phases. The time taken for the pulse wave to travel from the heart to the periphery is calculated and displayed as PTT.

**Display Functions:**
**Systolic and Diastolic Times:** Identifies key points in the PPG signal to calculate and display the durations of systolic and diastolic phases. It is important to note that to calculate **systolic times**, the function measures the difference in frequency between a

valley and the next peak. Meanwhile, for the **diastolic time**, the function calculates the difference in frequency between a peak and the next valley. Afterward, the function converts the results from the frequency domain to the time domain.

```
void display_systolic_diastolic(Signal *ppg) {
    int start = -1, end = -1, last = -1;

    for (int i = 1; i < ppg->num_values - 1; i++) {
        if (ppg->filtered_values[i] <= ppg->filtered_values[i - 1]
                && ppg->filtered_values[i] <= ppg->filtered_values[i + 1]
                && ppg->filtered_values[i] < calculate_filtered_mean(ppg)) {
            start = i;
            break;
        }
    }

    for (int i = start + 1; i < ppg->num_values - 1; i++) {
        if (ppg->filtered_values[i] >= ppg->filtered_values[i - 1]
                && ppg->filtered_values[i] >= ppg->filtered_values[i + 1]
                && ppg->filtered_values[i] > calculate_filtered_mean(ppg)) {
            end = i;
            break;
        }
    }

    for (int i = end + 10; i < ppg->num_values - 1; i++) {
        if (ppg->filtered_values[i] <= ppg->filtered_values[i - 1]
                && ppg->filtered_values[i] <= ppg->filtered_values[i + 1]
                && ppg->filtered_values[i] < calculate_filtered_mean(ppg)) {
            last = i;
            break;
        }
    }
    if (start != -1 && end != -1 && last != -1) {
        int sampling;
        if (ppg->window_sampling != 0)
            sampling = ppg->sampling / ppg->window_sampling;
        else
            sampling = ppg->sampling;
        float systolic_time = (end - start) * (1.0 / sampling); // Durata sistolica
        float diastolic_time = (last - end) * (1.0 / sampling); // Durata diastolica

        char systolic_msg[30], diastolic_msg[30];
        ssd1306_FillRectangle(0, 0, 128, 29, Black);
        sprintf(systolic_msg, "SysT: %.4f s", systolic_time);
        sprintf(diastolic_msg, "DiaT: %.4f s", diastolic_time);

        ssd1306_SetCursor(0, 0);
        ssd1306_WriteString(systolic_msg, Font_7x10, White);
        ssd1306_SetCursor(0, 10);
        ssd1306_WriteString(diastolic_msg, Font_7x10, White);
        ssd1306_UpdateScreen(&hi2c3);
    }
}
```

**Pulse Arrival Time:** Calculates and displays the PAT using peaks in the ECG and PPG signals.

```
void display_pulse_arrival_time(Signal *ecg, Signal *ppg) {
    int ecg_peak = -1, ppg_peak = -1;

    for (int i = 1; i < ecg->num_values - 1; i++) {
        if (ecg->filtered_values[i] >= ecg->filtered_values[i - 1]
                && ecg->filtered_values[i] >= ecg->filtered_values[i + 1]) {
            ecg_peak = i;
            break;
        }
    }
    for (int i = 1; i < ppg->num_values - 1; i++) {
        if (ppg->filtered_values[i] >= ppg->filtered_values[i - 1]
                && ppg->filtered_values[i] >= ppg->filtered_values[i + 1]
                && i > ecg_peak) {
            ppg_peak = i;
            break;
        }
    }
    if (ecg_peak != -1 && ppg_peak != -1) {
        int sampling = ppg->sampling / ppg->window_sampling;

        float pat = abs(ppg_peak - ecg_peak) * (1.0 / sampling);

        char pat_msg[30];
        sprintf(pat_msg, "PAT: %.2fms", pat * 1000);

        ssd1306_SetCursor(0, 20);
        ssd1306_WriteString(pat_msg, Font_7x10, White);
        ssd1306_UpdateScreen(&hi2c3);
    }
}
```

**Graph Updates:** Updates heart rate and HRV graphs on the OLED. The HRV graph is commented out to save space, but the heart rate graph provides real-time visual feedback.

```
void update_hr_hrv_graphs(Graph *hr_graph, Graph *hrv_graph, int new_hr,
        float new_hrv) {
    char msg[50];
    add_value(hr_graph, new_hr);
    add_value(hrv_graph, new_hrv);
    ssd1306_FillRectangle(0, 30, 128, 40, Black);
    sprintf(msg, "HR:  %d", new_hr);
    ssd1306_SetCursor(0, 30);
    ssd1306_WriteString(msg, Font_7x10, White);
    draw_graph(hr_graph, &hi2c3, 0, 40, 128, 24, 0, 150);

    /*
    ssd1306_SetCursor(0, 30);
    sprintf(msg, "HRV");
    ssd1306_WriteString(msg, Font_7x10, White);
    draw_graph(hrv_graph, &hi2c3, 0, 40, 128, 10, 0, 150);
    */

    ssd1306_UpdateScreen(&hi2c3);
}
```

Figure 61: HR graph and HRV graph (commented out)

## 7.6 Switching Modes

```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == GPIO_PIN_13 && debounce_in_progress == 0) {
        debounce_in_progress = 1;

        // Gestione del cambio di modalità
        current_mode = (current_mode + 1) % 3;
        switchMode(current_mode);

        if (current_mode != MODE_HOME) {
            HAL_TIM_Base_Start(&htim2);
            HAL_ADC_Start_DMA(&hadc2, (uint32_t*) data_out, ADC_NUM_CHANNEL);
            HAL_GPIO_WritePin(SDN_GPIO_Port, SDN_Pin, GPIO_PIN_SET);
        } else {
            timer3_counter = 0;
            HAL_TIM_Base_Stop(&htim2);
            HAL_ADC_Stop_DMA(&hadc2);
            HAL_GPIO_WritePin(SDN_GPIO_Port, SDN_Pin, GPIO_PIN_RESET);
            __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
        }

        // Disabilita l'EXTI per il debounce
        HAL_NVIC_DisableIRQ(EXTI15_10_IRQn);

        // Esegui un semplice ritardo software (busy-wait)
        for (uint32_t i = 0; i < debounce_delay; i++) {
            __asm("nop");
            // No Operation (NOP) per ritardo passivo
        }

        // Riabilita l'EXTI dopo il ritardo
        HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

        debounce_in_progress = 0; // Reset del flag di debounce
    }
}
```

Figure 62: Button Callback

The **GPIO EXTI Callback** function handles the input from the button connected to the microcontroller. When the button is pressed, the function begins with debounce logic to prevent multiple readings due to button bounce. Subsequently, it manages the application's mode switching by cyclically incrementing the current mode variable among three modes. Depending on the current mode:

- If it's not Home Mode, it activates a timer for timed operations, starts the ADC in DMA mode to acquire data, and sets a control pin to activate the ECG sensor.
- If it's Home Mode, it stops the timer and ADC, resets the control pin to deactivate the sensor, and resets the timer.

The function also handles the external interrupt interface to avoid multiple handling during debounce. This approach ensures a responsive handling of button inputs, dynamically adapting the application's behavior, and maintaining stable system operation.

## 7.7 Handling Bad Readings

When a bad sensor contact is detected, the system employs specific handling mechanisms for both PPG and ECG signals to ensure data integrity and user notification.

For the PPG sensor, upon detecting a bad contact, the system flags it and immediately

stops PPG data collection, interrupting the data collection process. The green LED is turned on, providing a visual indication of the issue. Additionally, messages sent over UART don't display values.

Similarly, for ECG signals, the system follows a comparable procedure. Upon detecting a bad contact, it stops ECG data collection and doesn't send values in the UART messages.

8

# Power Management

Embedded developers have a significant responsibility in ensuring that the firmware can minimize the energy consumption of designed devices. This is crucial for extending battery life and improving the overall efficiency of the system. Modern microcontrollers provide developers with a variety of tools to achieve minimal energy usage.When we talk about "power disspation" we refer to the conversion of electrical energy into thermal energy in a component, device or system.

Cortex-M cores, widely used in many embedded systems, offer an “abstract” power management model. This model is further refined by silicon manufacturers to create their own power management schemes. Among these, the STM32L and STM32U families from STMicroelectronics have reached a very sophisticated level of implementation.

The STM32L and STM32U families are designed with advanced power-saving features that enable developers to create highly efficient and energy-conscious applications. These microcontrollers include multiple low-power modes, flexible clock management, and optimized peripheral operation to reduce energy consumption during both active and idle states.

Power dissipation occurs due to various factors. The first aspect that affects power consumption is the operating frequency. The higher the CPU frequency, the greater the power consumption, as more switching activity occurs within the transistors. Additionally, the complexity of the device significantly impacts energy consumption. The more peripherals and features our board provides, the more power is required to support them.
A well-designed firmware should, therefore, immediately disable any peripheral that becomes unnecessary to minimize energy usage. Efficient power management involves dynamically enabling and disabling peripherals based on current operational needs, ensuring that only essential components consume power at any given time. The integration of external systems (e.g. sensors and actuators) can significantly increase the overall power usage. Efficient management of these components is essential to ensure they are only active when needed.

As we studied in our Electronics and Physics courses, power dissipation is measured in Watts (W), and it can be calculated using Ohm's Law. Ohm's Law relates the voltage (V), current (I), and resistance (R) in an electrical circuit, and is given by the formulas:

$$V = I \cdot R$$

Using Ohm's Law, power (P) can be calculated as:

$$P = V \cdot I$$

Alternatively, by substituting Ohm's Law into the power formula, we can express power dissipation in terms of resistance and current:

$$P = I^2 \cdot R$$

Or in terms of voltage and resistance:

$$P = \frac{V^2}{R}$$

These formulas are fundamental for understanding and calculating the power dissipation in electronic circuits. By applying these principles, embedded developers can better design their systems to manage power efficiently, ensuring optimal performance and energy usage. For some I2C devices, the resistence varies depending on the operating condition of the peripheral.

In this chapter, we will explain how we managed these features in our project and which coices we made in order to manage the device's power efficiency as best as possible. It is important to note that information about power dissipation of each component (sensor) used are provided inside in their datasheets

## 8.1 Estimated Power Consumption

In the following paragraphs we're gonna make an estimatation of the power dissipation of the board and the sensors.

### 8.1.1 Board NUCLEO-474RE

The IDE STM32CubeMX provides a specific tool that allows the calculation of an estimated power consumption for the board being used, which in our case is the NUCLEO-G474RE. This tool is known as the Power Consumption Calculator **(PCC)** and it enables developers to model and simulate the power usage of their applications, taking into account various operational states and peripheral activities.

First of all we need to configure our devices, and all peripherrals.

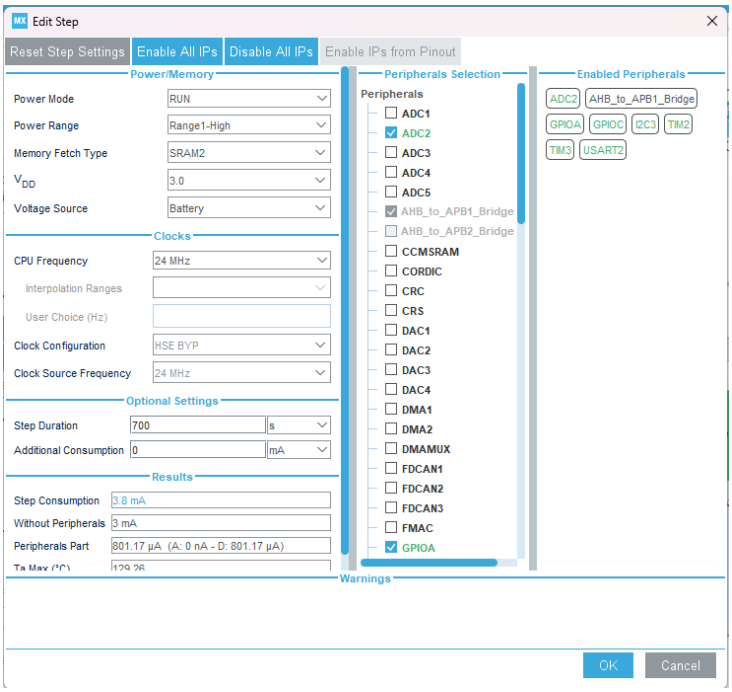



Figure 63: Edit Step

Below are the configuration parameters for the Power Consumption Calculator:

- **Power/Memory**
  - **Power Mode**: Select the power mode. In the image, `RUN` is selected, which represents the normal operating mode of the microcontroller.
  - **Power Range**: Set the power range. `Range1-High` is selected, allowing the microcontroller to operate at high performance.

- **Memory Fetch Type**: Select the type of memory to use. `SRAM2` is selected, which is a type of fast access memory.
- **$V_{DD}$**: Set the supply voltage. It is set to `3.0V`, which is a common voltage for many applications. It was not possibile to set 3.3V.
- **Voltage Source**: Select the voltage source. `Battery` is selected, indicating that the device is battery-powered.

- **Clocks**
  - **CPU Frequency**: Set the CPU frequency. it is set to `24 MHz`, balancing performance and power consumption.
- **Optional Settings**
  - **Step Duration**: Set the duration of the step. It is set to `700 s`, representing the time during which the microcontroller operates in this configuration.
  - **Additional Consumption**: Specify any additional consumption. It is set to `0 mA`, indicating that there are no additional consumptions beyond those calculated by the moment. (We'll make some considerations in the next paragraphs)

All these measurements are represented in the graph, which can be exported. The following is an example of such a graph:

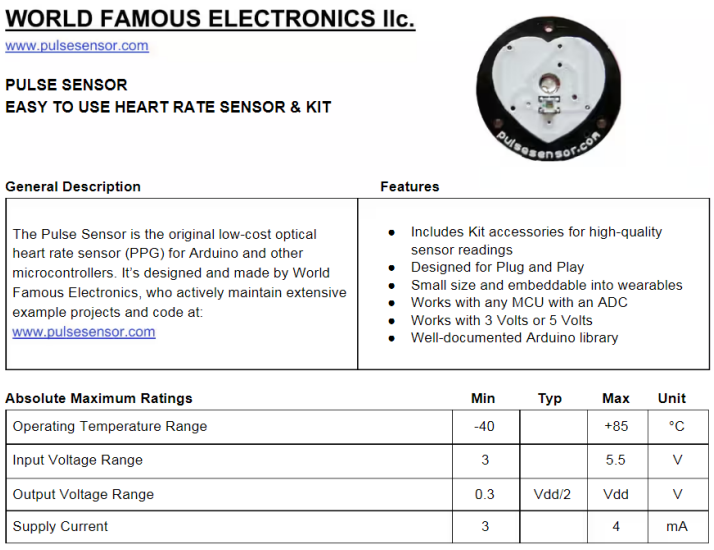

Figure 64: PCC graphical representation

As we can see the average cosumption has a value of 3.8mA -> 0.0038A, so we can calculate the Power Dissipation of the NUCLEO G474RE:

$$P_{G474RE} = V \cdot I = 3.0V \cdot 0.0039A = 0.0117W = 11.7mW$$

### 8.1.2 PPG sensor

PPG sensors use a light source (typically a LED) and a photodetector, making them optoelectronic devices. We referred to the relevant datasheet and found that the supply current required is approximately $4mA$ at a maximum operating temperature of $85°C$.



**WORLD FAMOUS ELECTRONICS llc.**
www.pulsesensor.com

**PULSE SENSOR**
**EASY TO USE HEART RATE SENSOR & KIT**



**General Description**

The Pulse Sensor is the original low-cost optical heart rate sensor (PPG) for Arduino and other microcontrollers. It's designed and made by World Famous Electronics, who actively maintain extensive example projects and code at:
www.pulsesensor.com

**Features**

- Includes Kit accessories for high-quality sensor readings
- Designed for Plug and Play
- Small size and embeddable into wearables
- Works with any MCU with an ADC
- Works with 3 Volts or 5 Volts
- Well-documented Arduino library

| Absolute Maximum Ratings | Min | Typ | Max | Unit |
|---|---|---|---|---|
| Operating Temperature Range | -40 | | +85 | °C |
| Input Voltage Range | 3 | | 5.5 | V |
| Output Voltage Range | 0.3 | Vdd/2 | Vdd | V |
| Supply Current | 3 | | 4 | mA |

Figure 65: PPG datasheet

Surely some of the necessary current is needed in order to activate the LED.

From this it was possible to calculate the power disspiation, considering also that the minimum power input was $3V$:

$$P_{PPG} = V \cdot I = 3.3V \cdot 0.004A = 0.0132W = 13.2mW$$

### 8.1.3 AD8232: ECG sensor

As mentioned before, by reading the documentation (datasheet) of the sensor, it was possible to understand that it operates at a maximum temperature of $70°C$ with a low supply current of $170\mu A$ (typical), we assume that it .

.

**AD8232** Data Sheet

| Parameter | Symbol | Test Conditions/ Comments | A Grade Min | A Grade Typ | A Grade Max | W Grade Min | W Grade Typ | W Grade Max | Unit |
|---|---|---|---|---|---|---|---|---|---|
| LOGIC INTERFACE | | | | | | | | | |
| Input Characteristics | | | | | | | | | |
| Input Voltage (AC/$\overline{DC}$ and FR) | | | | | | | | | |
| Low | $V_{IL}$ | | | 1.24 | | | 1.24 | | V |
| High | $V_{IH}$ | | | 1.35 | | | 1.35 | | V |
| Input Voltage ($\overline{SDN}$) | | | | | | | | | |
| Low | $V_{IL}$ | | | 2.1 | | | 2.1 | | V |
| High | $V_{IH}$ | | | 0.5 | | | 0.5 | | V |
| Output Characteristics | | LOD+ and LOD− terminals | | | | | | | |
| Output Voltage | | | | | | | | | |
| Low | $V_{OL}$ | | | 0.05 | | | 0.05 | | V |
| High | $V_{OH}$ | | | 2.95 | | | 2.95 | | V |
| SYSTEM SPECIFICATIONS | | | | | | | | | |
| Quiescent Supply Current | | | | 170 | 230 | | 170 | 230 | µA |
| | | $T_A$ = 0°C to 70°C | | 210 | | | | | µA |
| | | $T_{OPR}$ | | | | | | 270 | µA |
| Shutdown Current | | | | 40 | 500 | | 40 | 500 | nA |
| | | $T_A$ = 0°C to 70°C | | 100 | | | | | nA |
| | | $T_{OPR}$ | | | | | | 612 | nA |
| Supply Range | | | 2.0 | | 3.5 | 2.0 | | 3.5 | V |
| Specified Temperature Range | | | 0 | | +70 | −40 | | +105 | °C |
| Operational Temperature Range | | | −40 | | +85 | −40 | | +105 | °C |

Figure 66: ECG datasheet

From this we could calculate the Power disspation of the ECG sensor:

$$P_{ECG} = V \cdot I = 3.3V \cdot 0.00017A = 0.000561W = 0.561mW$$

### 8.1.4 SSD1306: OLED Display

From the documentation, we can note that with the contrast set to FFh (maximum level of contrast). the average current consumption is 430 µA and the masximum is 780 µA. Considering that the screen is only in part in use cause the OLED display has pixel unused completely turned off we can assume that the needed current is the aveaverage 430 µA.

| | | | | | | |
|---|---|---|---|---|---|---|
| $I_{CC}$ | $V_{CC}$ Supply Current<br>$V_{DD}$ = 2.8V, $V_{CC}$ = 12V,<br>$I_{REF}$ = 12.5uA<br>No loading, Display ON, All ON | Contrast = FFh | - | 430 | 780 | uA |
| $I_{DD}$ | $V_{DD}$ Supply Current<br>$V_{DD}$ = 2.8V, $V_{CC}$ = 12V,<br>$I_{REF}$ = 12.5uA<br>No loading, Display ON, All ON | | - | 50 | 150 | uA |

Figure 67: SSD1306 datasheet

So the Power consumption of the OLED is:

$$P_{OLED} = V \cdot I = 3.3V \cdot 0.000430A = 0.001419W = 1.419mW$$

### 8.1.5 DS3231: RTC sensor

In this case too, we consulted the datasheet of the DS3231 RTC. The datasheet provided essential information about the operating conditions and power requirements of the DS3231.

**ELECTRICAL CHARACTERISTICS**

($V_{CC}$ = 2.3V to 5.5V, $V_{CC}$ = Active Supply (see Table 1), $T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Typical values are at $V_{CC}$ = 3.3V, $V_{BAT}$ = 3.0V, and $T_A$ = +25°C, unless otherwise noted.) (Notes 1, 2)

| PARAMETER | SYMBOL | CONDITIONS | | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|---|
| Active Supply Current | $I_{CCA}$ | (Notes 3, 4) | $V_{CC}$ = 3.63V | | | 200 | µA |
| | | | $V_{CC}$ = 5.5V | | | 300 | |
| Standby Supply Current | $I_{CCS}$ | I²C bus inactive, 32kHz output on, SQW output off (Note 4) | $V_{CC}$ = 3.63V | | | 110 | µA |
| | | | $V_{CC}$ = 5.5V | | | 170 | |
| Temperature Conversion Current | $I_{CCSCONV}$ | I²C bus inactive, 32kHz output on, SQW output off | $V_{CC}$ = 3.63V | | | 575 | µA |
| | | | $V_{CC}$ = 5.5V | | | 650 | |

Figure 68: RTC datasheet

In order to understand the needed current, we referred to the field "Active Battery Current" in the datasheet. So the power consumption of the sensor is the following:

$$P_{RTC} = V \cdot I = 3.3V \cdot 0.0002A = 0.0006W = 0.66mW$$

### 8.1.6 LED

The power dissipated by the LEDs implemented in our project was calculated considering the resistor configurations. First of all let's consider the normal power disspiation of the LED:

$$P_{LEDRed} = V_f \cdot I = 1.8V \cdot 0.02mA = 0.066W = 66mW$$

$$P_{LEDGreen} = V_f \cdot I = 2.1V \cdot 0.02mA = 0.042W = 42mW$$

For the red LED, we used two resistors (one of 100 Ω and one of 330 Ω) in parallel. For the green LED, we used two resistors (both of 100 Ω) in parallel with one resistor (10 Ω) in series. Considering that the internal resistance of the LED without resistors was respectively:

$$R_{\text{LEDRed}} = \frac{V_f}{I} = \frac{1.8\,V}{0.02\,A} = 90\,\Omega$$

$$R_{\text{LEDGreen}} = \frac{V_f}{I} = \frac{2.1\,V}{0.02\,A} = 105\,\Omega$$

Therefore, the total resistance is respectively:

$$R_{\text{LEDRedtot}} = 75\,\Omega + 90\,\Omega = 165\,\Omega \quad \text{for the red LED}$$

$$R_{\text{LEDGreentot}} = 60\,\Omega + 105\,\Omega = 165\,\Omega \quad \text{for the green LED}$$

From this assumption, we can calculate the power dissipation for each LED, which is the same because the values of voltage and resistance are the same at this point:

$$P_{\text{LEDs}} = \frac{V^2}{R_{\text{eq}}} = \frac{3.3^2\,V}{165\,\Omega} = 0.1452\,W = 145.2mW \cdot 2 = 290.4mW$$

Considering the red LED uses Pulse Width Modulation (PWM), the power dissipation can be calculated as the average power over one period (5000 ms).

The instantaneous power dissipation $P(t)$ when the LED is on is given by:

$$P(t) = \frac{V^2}{R} = \frac{3.3^2\,V}{165\,\Omega} \approx 0.066\,W = 66\,\text{mW}$$

For a PWM period $T = 4$ ms with a duty cycle $D$, the average power dissipation $\overline{P}$ can be calculated as the integral of the power over one period:

$$\overline{P} = \frac{1}{T}\int_0^T P(t)\,dt$$

Since the power $P(t)$ is constant during the on-time and zero during the off-time, we have:

$$P(t) = \begin{cases} P & \text{for } 0 \le t < D \cdot T \\ 0 & \text{for } D \cdot T \le t < T \end{cases}$$

Thus, the average power dissipation is:

$$\overline{P} = \frac{1}{T}\left(\int_0^{D\cdot T} P\,dt + \int_{D\cdot T}^{T} 0\,dt\right)$$

Simplifying the integral:

$$\overline{P} = \frac{1}{T}\left(P \cdot D \cdot T\right) = P \cdot D$$

The energy dissipated over the period $T$ is then:

$$E_{\text{LED-Fade}} = \overline{P} \cdot T = (66\,\text{mW} \cdot D) \cdot 0.004\,\text{s}$$

Considering the red LED uses Pulse Width Modulation (PWM), with a period of 4 ms, the power dissipation can be calculated as the average power over one period.
The instantaneous power dissipation $P(t)$ when the LED is on is given by:

$$P = \frac{V^2}{R} = \frac{3.3^2\,V}{165\,\Omega} \approx 0.066\,W = 66\,\text{mW}$$

For a PWM period $T = 4$ ms with an unknown duty cycle $D$, the average power dissipation $\overline{P}$ is:

$$\overline{P} = P \cdot D = 66\,\text{mW} \cdot D$$

The energy dissipated over the period $T$ is then:

$$E_{\text{LED-Fade}} = \overline{P} \cdot T = 66\,\text{mW} \cdot D \cdot 4\,\text{ms}$$

Converting milliseconds to hours, we get:

$$E_{\text{LED-Fade}} = 66\,\text{mW} \cdot D \cdot 4 \times 10^{-3}\,\text{s} \times \frac{1\,\text{hour}}{3600\,\text{s}} = 66\,\text{mW} \cdot D \cdot \frac{4}{3600}\,\text{hours}$$

$$E_{\text{LED-Fade}} = 66\,\text{mW} \cdot D \cdot 0.0011\,\text{hours} = 0.0726\,\text{mWh} \cdot D$$

This calculation shows that the energy dissipated by the LED using PWM over a period of 4 ms is 0.0726 mWh $\cdot$ $D$, where $D$ is the duty cycle.

## 8.2 Final Calculus:

After evaluating each component, let's make a summary of all the power dissipated by each component:

$$P_{Tot} = P_{G474RE} + P_{PPG} + P_{ECG} + P_{OLED} + P_{RTC}$$

$$P_{Tot} = 11.7mW + 13.2mW + 0.561mW + 1.419mW + 0.66mW = 27.54mW$$

So in conclusion considering that $P_{tot}$ in 1 hour is 27.54mWh :

- Power consumption of the board and peripherrals: 27.54mWh
- RED Led: 0.0726mWh $\cdot$ D per duty cycle considering the synchronization with the signal that is acquiring the ADC
- GREEN Led (when it is turned on) : 145.2mWh depending to the time it is turned on.

## 8.3 PCC: Adding calculus to the tool

As mentioned earlier, there was a field initially set to zero—the counter for the consumption of the external peripherals. Now that we know the effective consumption of the peripherals, it was possible to add this value to the step. For semplicity and for projectual choices (because maybe not too much accurate) the calculus of a LED power dissipation has not been inserted.

To accurately account for the power consumption of all components, the external peripherals' consumption values were integrated into the power calculation tool (See Additional Consumption field) . This involved updating the step configuration to reflect the actual power usage, ensuring a more precise estimation of the total power consumption.

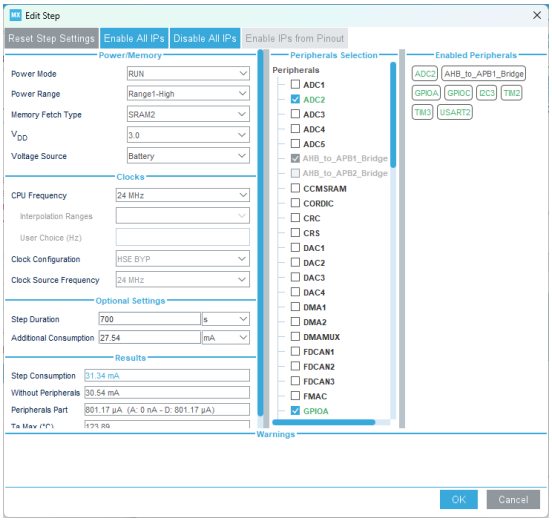



Figure 69: PCC after calculus

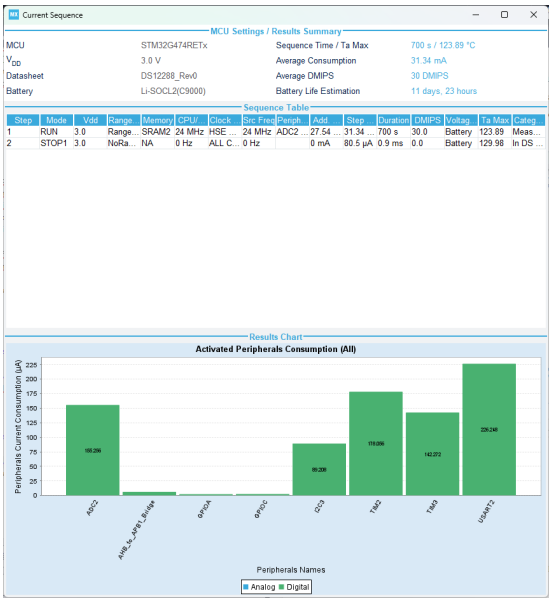



Figure 70: PCC after calculus

It is evident that the power consumption has increased. Initially, the battery life could last more than a month. After the insertion of the peripherals, the duration (for a Li-SOCL2 (A3400) battery) was reduced to 4 days. Consequently, I changed the battery to a Li-SOCL2 (C9000) to achieve a duration of 11 days, which is a very good result.

9

# Final Conclusions

In conclusion, all project specifications have been successfully met. However, we faced significant challenges during the phase of detecting heartbeats using the PPG sensor, mainly due to its high sensitivity to light. This sensitivity caused disturbances in the signal, making it difficult to distinguish between valid signals and environmental interferences. Additionally, we had to experiment with various parameters to find a filter suitable for different user profiles.

Despite these challenges, the final considerations are positive as we have achieved our set objectives. Looking ahead, we could enhance the device by replacing the PPG sensor with a more advanced one and improving the display to create a more comfortable and notably more accurate smartwatch.

These enhancements could not only increase the accuracy of physiological measurements but also make the device more appealing for everyday use.