Course of Embedded Systems for E-Health

LM-32 INFORMATION ENGINEERING FOR DIGITAL MEDICINE

Smart Medical Device Health parameters monitor

APICELLA SALVATORE: 0623200009

BOVE SIMONE: 0623200004

CASCONE MARIA

a.a. 2022/2023

Contents

1 Introduction **3**

2 Solution **4**
2.1 The Problem 4
2.1.1 Methods 4
2.2 The Context 5
2.3 The Device 5
2.4 Measurement Locations 6
2.5 Graphical Interfaces 6
2.6 Thresholds 7
2.6.1 Fever on the Wrist 7
2.6.2 Tachycardia 7
2.6.3 Oxygen Saturation 7

3 Design Choices **9**
3.1 Devices and Protocols Available 11
3.1.1 KY028 11
3.1.2 MAX32664 12
3.1.3 SSD1306 16
3.1.4 DS1307 17
3.1.5 Buzzer 18
3.1.6 LED 19
3.2 Our Development Environment 20
3.3 .ioc Setup 20
3.3.1 Clock 21
3.3.2 I2C 21
3.3.3 UART 22
3.3.4 TIM 22
3.3.5 ADC 22
3.3.6 PINOUT 22

4 Hardware Architecture **24**

5 Software Architecture **28**
5.1 action.h . . . 32
5.2 main.h . . . 32
5.2.1 Flag . . . 32
5.2.2 Disease . . . 33
5.2.3 Measure . . . 33

6 Software Interfaces **34**
6.1 UART2 (Debug) . . . 35
6.2 TIM1 for PWM . . . 35
6.3 TIM10 for generating interrupts every second . . . 36
6.4 TIM11 for generating interrupts every millisecond . . . 36
6.5 ADC for reading a temperature sensor . . . 37
6.6 I2C . . . 37

7 Software Protocols (designed by us) **39**
7.1 Initialization . . . 39
7.2 TIM 10 . . . 41
7.2.1 Measurement . . . 41
7.2.2 10 Measurements made . . . 42
7.2.3 Exception handling . . . 44
7.2.4 Finger not detect . . . 45
7.2.5 Raise some alert . . . 45
7.3 TIM 11 . . . 47

8 Power Dissipation **49**
8.1 What is? . . . 49
8.2 Power Calculation . . . 50
8.2.1 NUCLEO-F401RE . . . 50
8.2.2 KY-028 . . . 51
8.2.3 MAX32664 . . . 51
8.2.4 DS1307 . . . 52
8.2.5 SSD1306 . . . 52
8.2.6 LED . . . 52
8.2.7 Buzzer . . . 53
8.2.8 Final Calculation . . . 54

Chapter 1

Introduction

In the medical field, it is increasingly important to rely on machines for patient monitoring. For obvious logistical reasons, it is impossible for a doctor to monitor a patient's parameters throughout the entire day. Therefore, it becomes necessary to use devices that can continuously monitor the patient and record their vital parameters.

There are various devices that serve this purpose, each with a specific goal. For example, the Holter monitor is a medical device that allows for continuous monitoring of a patient's cardiac parameters through an electrocardiogram. However, it is not intended for daily use by the patient, as the collected data is later analyzed by the doctor who installed the device.

Another commonly used device is the Smart Watch, which is widely available on the market and often has an affordable cost. It allows the user to collect data about themselves and conveniently analyze it from their smartphone.

Nowadays, there are many different options for collecting and analyzing patients' vital parameters, and the quality of measurements and the way data is processed can vary.

In this report, we will focus on how to build one of these devices, addressing both the hardware and software design aspects.

Chapter 2

Solution

Within the course of Embedded Systems for E-Health, we have acquired skills that give us the ability to utilize various technologies for embedded systems, including those particularly useful for analyzing data from sensors. We have also gained an understanding of how sensors are constructed and developed awareness of their usage.

2.1 The Problem

The problem presented by the professors is as follows:

Design and realize the prototype of a smart device for health monitoring based on the measurement of:

- **Body temperature**
- **Heart rate**
- **Blood oxygenation**

2.1.1 Methods

We were also required to consider specific technical aspects, namely:

- Measure, aggregate information, set thresholds
- Display the main information on a screen
- Send detailed information about each measurement to a PC through USB, associated with a timestamp

Additionally, we were also asked to:

- Interact with the patient: provide suggestions in case of values outside the healthy range
- Estimate the power consumption of the device

2.2 The Context

Based on the requirements set by the professors, we had to undertake thorough planning. However, before doing so, we considered the context in which our device would be used.

Firstly, the device must be designed to interact with the patient even in the absence of a doctor. However, considering the device's ability to communicate via USB, we envision its use within a healthcare facility such as a hospital or clinic, if not a doctor's practice.

The device needs to communicate with the patient in the clearest possible manner, taking into account that the patient is not a technician and should not be concerned with the thresholds that trigger alert messages; that is the responsibility of specialized personnel.

The device is designed to operate continuously and thus without sleep mode.

2.3 The Device

The device we will create is an embedded system, for which we will design both the circuit and the software. It will be based on the NUCLEO-F401RE board and will incorporate an external clock to establish the time of each measurement. It will include sensors for heart rate, blood oxygenation, and temperature. Additionally, it will feature a screen, a LED and a buzzer for user interaction.

The idea is to create a device that utilizes simple and iconic graphics to indicate vital parameters to the user. It will alert the user with an audible signal and display a screen detailing the issue. Additionally, it may provide potential solutions for the user.

The circuit will be enclosed in a small case to prevent the user from coming into contact with unnecessary components, focusing their attention solely on

the areas of interaction, namely the device's sensors and actuators.

2.4 Measurement Locations

A fundamental aspect of our device is the appropriate selection of measurement locations. For example, oxygenation devices tend to work best when measurements are taken on a fingertip, while temperature measurements are not ideally taken in this peripheral area due to the significant difference in temperature from the standard 36°C.

2.5 Graphical Interfaces

As previously mentioned, we believe it is of utmost importance to provide the user with a graphical interface that is as simple as possible. Therefore, we have opted for a GUI that constantly displays all measurements on the screen. The display will be divided into three rows, with an icon representing a thermometer, a heart, and oxygen, followed by the respective parameter values and units of measurement. In case a measurement is not available, dashes will be displayed instead of the value.

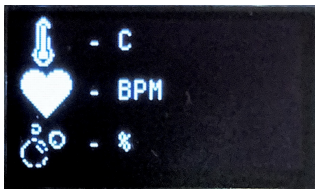

Figure 2.1: Graphical Interfaces - Default

We also need to handle cases where the user is not correctly positioned on the sensor for an extended period and display a screen to prompt them to position themselves properly.

Figure 2.2: Graphical Interfaces - Finger not detected

Additionally, we must manage emergency cases, and we have considered two approaches: one for the first occurrence of an alarm and another for subsequent occurrences.

- First occurrence: The buzzer is activated to signal the alert, and an emergency screen is displayed briefly, clearly indicating the problem to the user. In the case of tachycardia, a breathing mode is activated, guiding the user through screens to encourage deep breathing.
- Subsequent occurrences: A triangle-shaped alert icon will be displayed next to the parameter to indicate that it is outside the normal range.

In addition to these graphical interfaces that show processed data to the users, it is also possible to read raw values through the terminal, which are sent by the board for more accurate monitoring.

2.6 Thresholds

To ensure the device functions correctly, it will be necessary to calibrate all the sensors accurately. Furthermore, thorough research on the appropriate thresholds for triggering alarms is necessary based on medical literature.

2.6.1 Fever on the Wrist

After reviewing relevant literature, we found a study conducted to simplify the diagnosis of COVID-19 [1], which identified a threshold at which a fever symptom can be detected on the wrist, at 36.15°C.

2.6.2 Tachycardia

Firstly, it is important to note that heart rate is monitored while the patient is at rest. We found a study that analyzed a large sample of people, identifying a threshold for tachycardia at 90 BPM at rest [2].

2.6.3 Oxygen Saturation

While studying various articles, we did not find general articles on appropriate saturation levels. However, we found many studies that indicated thresholds to aid in the diagnosis of bronchitis or COVID-19. For our project, we chose to refer to studies regarding COVID-19, which identified the correct threshold at 95% saturation [3].

- Temperature: Above 36.15°C on the wrist
- Heart Rate: Above 90 BPM
- Oxygen Saturation: Below 95%

Chapter 3

Design Choices

First of all, after reading the prompt, we started a design phase, which was essential to have a clear understanding from the beginning and avoid significant modifications later on.

We had to make some design choices, which required us to find an optimal configuration to make the best use of the available resources. We decided to represent the natural execution cycle of our software on the device through a state diagram (Figure 3) to understand what would be useful and what wouldn't.

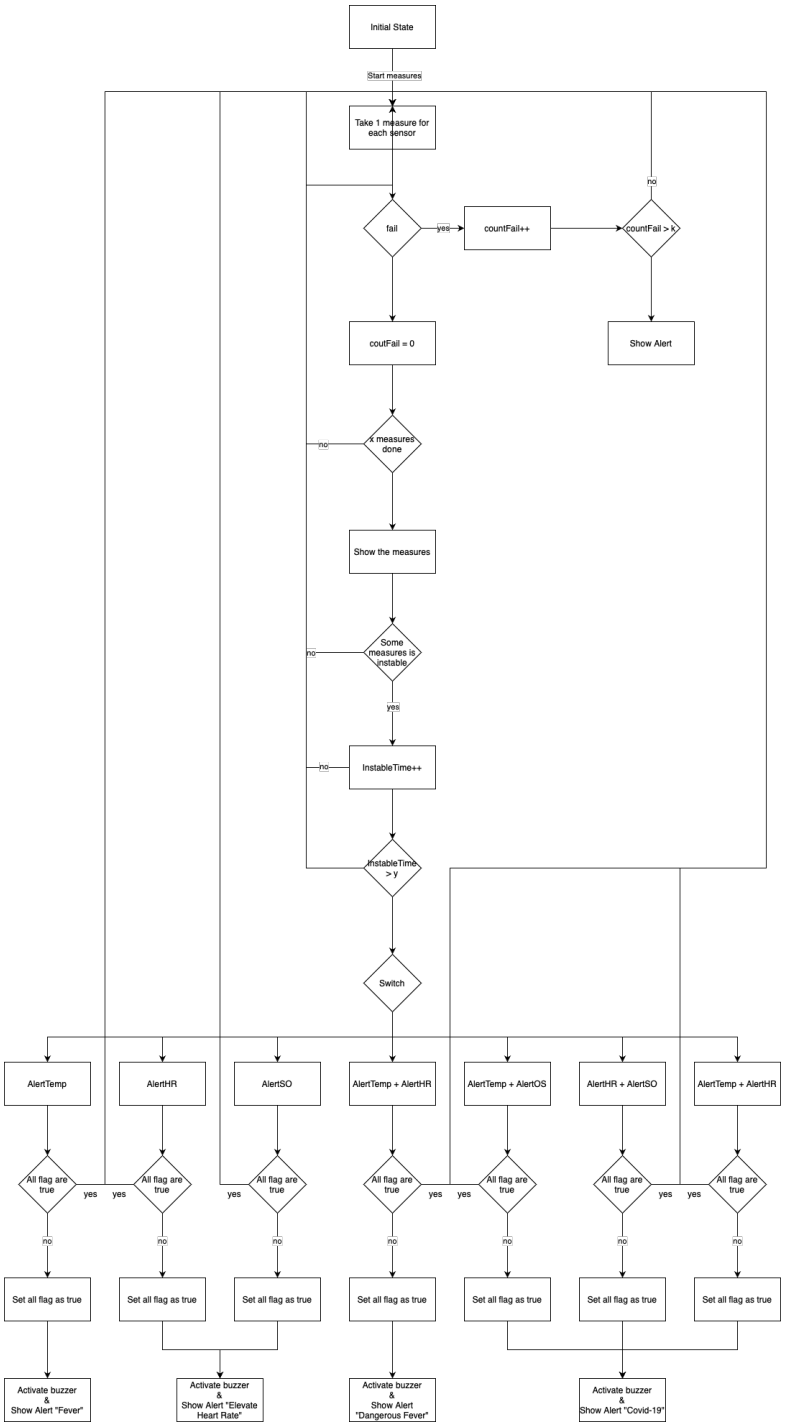

Figure 3.1: State Diagram

3.1 Devices and Protocols Available

We then compiled a list of things needed for the implementation, particularly highlighting the control technologies involved.

For the realization of our project, we will be using:

- 1 KY028 temperature sensor, which operates using **ADC**.
- 1 MAX32664 heart rate and saturation sensor, which operates using **I2C**.
- 1 SSD1306 OLED screen, which operates using **I2C**.
- 1 DS1307 external clock, which operates using **I2C**.
- 1 passive buzzer to be controlled using **PWM**.
- 1 external LED to be controlled using **PWM**.

In addition to these peripherals, we have also decided to use the UART protocol to obtain useful data for debugging purposes, as well as the internal timers of the board to manage certain events that may occur. Now let's analyze how our sensors work to fully understand their functioning.

3.1.1 KY028

The KY-028 Temperature Sensor module is a sensor that can measure temperature using a thermistor. The thermistor is a type of resistor whose resistance changes with temperature.

Resistance is the measure of how much an object obstructs the flow of current.

$$R = \frac{V}{I} \tag{3.1}$$

Resistance depends on the geometry and material of the object. It is also influenced by the resistivity of the medium, which in turn depends on the geometric shape, temperature, and medium.

$$R = \rho \cdot \frac{L}{S} \tag{3.2}$$

In our specific case, the sensor uses an NTC thermistor, which means we have a resistance that increases as the temperature decreases. The operation of a thermistor is based on the principle that temperature influences the mobility

of electrons in the semiconductor material. At higher temperatures, thermal energy provides electrons with greater energy, allowing them to move more easily and increasing the electrical conductivity of the material.

Remember that resistivity is the inverse of conductivity.

$$\rho = \frac{1}{\sigma} \tag{3.3}$$

To associate temperature with resistance, you can refer to the conversion table provided in the specific datasheet of the thermistor.

However, the sensor is also equipped with a potentiometer that allows calibrating the sensitivity of the sensor and, most importantly, setting a threshold for the digital output pin. Specifically, the potentiometer has a resistance of 100000Ω.

To use the sensor, it needs to be connected with the positive pin (+) to the 3.3V, the negative pin (-) to the ground (GND), and the analog pin to the ADC pin of our NUCLEO-F401RE (In our context, the digital pin remains unconnected).

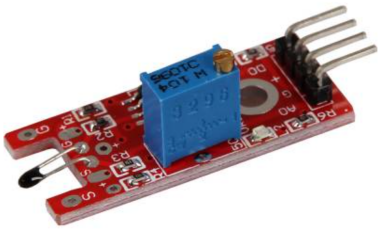

Figure 3.2: KY028

Calibration

Device KY028, as previously reported, has a potentiometer to improve its sensitivity, and taking advantage of this factor, we had the sensor temperature stabilized at a known measurement and adjusted the potentiometer accordingly.

3.1.2 MAX32664

The MAX32664 sensor is a biometric sensor based on I2C that utilizes two integrated chips:

- MAX32664A Biometric Sensor Hub
- MAX30101 Optical Sensor

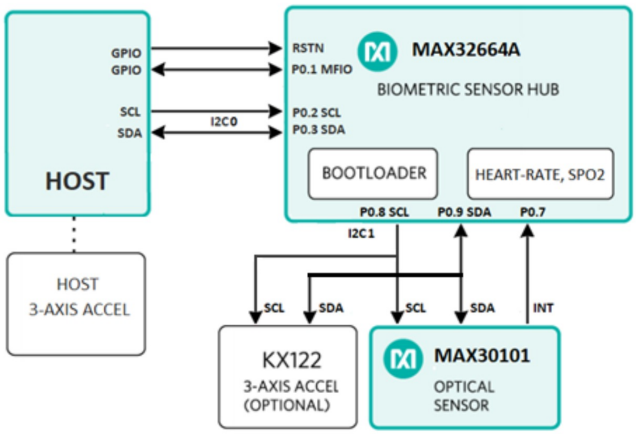

Figure 3.3: Architecture Diagram of MAX32664

The first device is a tiny Cortex-M4 processor capable of handling all the data provided by the second device, which is responsible for detection. The MAX32664 is used as the controller of the sensor hub and allows processed or unprocessed data to be sent to a host device. In addition to the I2C pins, the device has two GPIO pins:

- RSTN (reset)
- MFIO (multi-function input/output)

These pins are useful for switching between the two modes that the device has:

- Bootloader mode
- Application mode

The first mode is used to load or update the firmware of the device, while the second mode is the normal operating mode in which data is detected. In our case, we are using the Application Mode, and to access this mode, the following sequence of steps needs to be followed:

1. Set the RSTN pin to 0 for 10ms.
2. Set the MFIO pin to 1.
3. Set the RSTN pin to 1.

The sensor allows for the calculation of both blood oxygen saturation values and heart rate. It utilizes pulse oximetry technique in the case of blood oxygenation, while optical pulses are used in the case of heart rate measurement.

Pulse oximetry: SpO2

Pulse oximetry is a non-invasive technique used to calculate the oxygen saturation in the blood. This technique relies on a pulse oximeter that combines infrared and red light to determine the amount of oxygen bound to hemoglobin in the blood.

> The pulse oximetry sensor is based on the principle that hemoglobin in the blood behaves differently in the presence and absence of oxygen. Oxygenated hemoglobin (HbO2) absorbs light differently than deoxygenated hemoglobin (Hb) at specific wavelengths.

The main components used in a pulse oximeter are:

- LEDs (Light-Emitting Diodes)
    1. Light Emission: LEDs emit light through the tissue. The emitted light can be of different types, but the main wavelengths used are typically around 660 nm (red light) and 940 nm (infrared light). These wavelengths are chosen because hemoglobin has different absorption peaks at these wavelengths.
    2. Light Absorption: The emitted light passes through the tissue and is absorbed by hemoglobin in the blood. Oxygenated hemoglobin (HbO2) absorbs more infrared light, while deoxygenated hemoglobin (Hb) absorbs more red light. This is because hemoglobin exhibits different absorption behavior depending on its oxygenation state.
- Photodiodes
    1. Signal Detection: Photodiodes detect the light that is either reflected or transmitted through the tissue. The amount of light detected by the photodiodes depends on the level of light absorption by hemoglobin in the blood.
- Signal Processing Electronics
    1. Signal Processing: The signal detected by the photodiodes is processed by electronic circuits within the sensor. These circuits amplify the signal to make variations in light intensity more evident.

- Calculation of SpO2 values.

Optical Pulse Sensor: Heart Rate

The optical pulse sensor, it uses a photoplethysmography (PPG) technology, which is based on measuring variations in light reflected or transmitted through body tissues. When the heart pumps blood through the body, there is a change in the volume of blood within the blood vessels. This volume change causes variations in the amount of light that is absorbed or reflected by the tissues. The optical sensor captures these variations and converts them into electrical signals.

The workflow of this technology is:

- Light Emission:
- Light Absorption and Reflection
- Detection of reflected or transmitted light
- Signal processing
- Calculate Heart Rate

The first step the sensor takes is to emit light with wavelengths that are well absorbed by biological tissue. So, the emitted light is then absorbed by the tissues, and some of it is reflected. This process depends on the volume of blood present in the blood vessels.
During ***systole*** (contraction) of the heart, the volume increases, resulting in greater light absorption.
During ***diastole*** (relaxation) of the heart, the blood volume decreases, leading to a decrease in the amount of light absorbed.
The reflected light is then captured by a ***photodiode***, which converts the light into an electrical signal proportional to the detected light intensity. The final step is to combine these data and calculate the heart rate value.

Figure 3.4: MAX32664

A common component between the two sensors is the ***photodiode***.

> Photodiode is an electronic device that converts light into an electric current. It operates based on the photoelectric effect, a phenomenon in which incident light photons on a material can release electrons, generating an electric current proportional to the intensity of the light. The photodiode is very similar to a diode but is modified to be sensitive to light. Like a diode, it has a p-doped region and an n-doped region, with a greater distribution for the p-doped region. When light strikes the photodiode, photons are absorbed by the semiconductor material, and the energy of the photons excites electrons, generating a measurable current.

The accuracy of a photodiode depends on the wavelength of the photons. Photodiodes can be optimized to respond to specific wavelength ranges, such as detecting infrared light.

3.1.3 SSD1306

The SSD1306 OLED is a type of organic light-emitting diode (OLED) display that utilizes the SSD1306 controller. It operates by using an array of small organic diodes that emit light when a voltage is applied to them.

An OLED diode consists of two organic materials, an emissive layer and a transport layer. The emissive layer emits photons when excited by an electric current, while the transport layer allows the passage of electrons to the emissive layer. The electrons in the emissive layer combine with the holes (electron-absent charge carriers) and create excitations that release energy in the form of photons, which is visible light.

The display matrix is a 128×64 pixel array, and each pixel can be controlled independently to vary brightness and create images or text. The display is monochromatic, allowing only the display of white (illuminated) and black (off) colors.

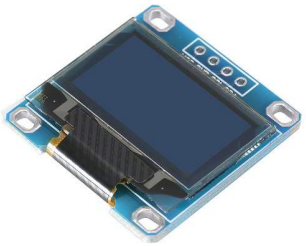

Figure 3.5: SSD1306

In terms of communication, the OLED display utilizes the I2C communication protocol.

When data is sent to the SSD1306 OLED transducer, the SSD1306 controller interprets the information and activates the corresponding pixels in the emission layer of the OLED display. This way, the image or characters are displayed on the OLED screen.

3.1.4 DS1307

The DS1307 utilizes a combination of digital circuits and a quartz crystal to generate a stable and precise clock signal. The module is powered by a button battery, which keeps the clock functioning even when the main power is interrupted.

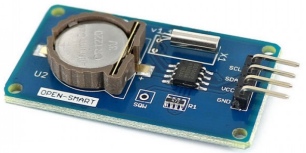

Figure 3.6: DS1307_RTC

Here is how the DS1307 works in detail:

Oscillator: The DS1307 module includes an internal oscillator that uses a quartz crystal to generate a clock signal. The quartz ensures a stable and

precise frequency for timekeeping.

Calendar: The DS1307 keeps track of time, date, and calendar information. It supports both 12-hour and 24-hour formats for displaying the time and can handle leap years as well.

Registers: The DS1307 has a set of registers to store time and calendar information. The registers include hours, minutes, seconds, day of the week, date, month, and year. These registers can be read from or written to through an I2C serial communication interface.

Interrupts: The DS1307 can generate periodic interrupts or respond to specific conditions. These interrupts can be used to trigger system events or perform actions based on programmed timings.

Battery Backup: The DS1307 module is equipped with a connector for a button battery (typically a lithium battery) that provides backup power. When the main power is interrupted, the DS1307 continues to function using the battery power to keep the clock running and preserve calendar data.

Communication: The DS1307 uses the I2C serial interface to communicate with the hosting microcontroller or microprocessor. This serial interface requires only two data lines (SDA and SCL), simplifying interfacing with other devices.

3.1.5 Buzzer

A passive buzzer is an electronic device used to produce sounds or acoustic signals when an electric current is applied to it. It consists of a diaphragm, a movable coil, a permanent magnet, and a system of electrical contacts.

Figure 3.7: Passive Buzzer

When an electrical signal is applied to the movable coil, a magnetic field is

generated that interacts with the permanent magnet. This causes an oscillatory movement of the movable coil, which in turn vibrates the diaphragm. The vibration of the diaphragm produces sound waves in the surrounding air, generating the desired sound.

A passive buzzer requires an external signal to generate sound and does not have an internal control circuit. To control the amplitude and frequency of the produced sound, a technique called pulse width modulation (PWM) can be used.

Pulse width modulation (PWM) is a technique in which the signal is modulated between two states: a high level and a low level, at regular intervals. By modulating the ratio between the period in which the signal is high compared to the total period, it is possible to control the effective amplitude of the electrical signal.

Using PWM to drive a passive buzzer allows for controlling the volume of the produced sound by adjusting the duty cycle ratio of the PWM signal. Additionally, it enables generating a wide range of tones and frequencies, offering greater flexibility in generating the desired sounds.

In summary, a passive buzzer is a device that produces sounds when an electric current is applied to it. The vibration of the diaphragm generates the desired sound. By using pulse width modulation (PWM) technique, it is possible to precisely and flexibly control the amplitude and frequency of the produced sound.

3.1.6 LED

LEDs (Light-Emitting Diodes) are electronic devices that emit light when an electric current passes through them. They are commonly used for lighting, visual indicators, and display screens.

Figure 3.8: Led

The basic operation of an LED is based on the principle of light emission through the electroluminescent process. Here's how an LED works:

- LED Structure: An LED consists of two layers of semiconductor materials: the P-type (positive) semiconductor material and the N-type (negative) semiconductor material. The junction between these two layers is called the P-N junction.
- Applying a Voltage: When a voltage is applied to the P-N junction, an electric field is created within the LED. This electric field pushes electrons from the regions with high electron concentration (N) towards the regions with high hole concentration (P).

It is important to note that LEDs are polarized diodes, which means they operate in only one direction. The correct polarity must be observed when applying voltage to the LED to avoid damage.

To control an LED, the correct voltage and current must be applied to the LED itself. This can be achieved by using an appropriate series resistor to limit the current passing through the LED and a controlled power source, such as a microcontroller or control circuit.

3.2 Our Development Environment

To program the NUCLEO-F041RE board, we chose to use the "STM32CubeIDE" programming environment as it is fully compatible with the boards and the most familiar environment, having been used during the course.

Within this environment, we can find useful debugging tools, as well as a set of tools to modify the board's configuration files effectively.

For collaborative software development, we utilized a GitHub repository.

3.3 .ioc Setup

The .ioc file is of fundamental importance for the operation of our board as it allows us to configure all the environment parameters. We can specify the behavior for each implemented protocol, such as how the board should behave and how to configure its physical connections.

After considering the sensors and actuators to be used, we concluded that we need to implement the following technologies:

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

3.3.1 Clock

After thorough analysis and considering that we want to measure vital parameters over a time period in the order of seconds, we don't require a particularly high clock speed.

Considering this requirement, the fact that a lower clock consumes less energy, and the potential for greater system stability due to reduced noise or interference, we chose the lowest clock frequency offered by our board: 16MHz.

We set the system clock to 16MHz by selecting the HSI line from the multiplexer and setting all subsequent prescalers to "/1" so that all clock outputs have the same frequency of 16MHz.

3.3.2 I2C

Since three of the peripherals we use, namely the MAX32664 sensor, the SSD1306 OLED screen, and the external DS1307 clock, utilize the I2C protocol, we implemented this protocol within the project. Among the available I2C options, we chose I2C1.

Before deciding how many I2C buses to use, we made some considerations. Using a single bus simplifies the circuit and code, but there is a risk of conflicts if multiple peripherals want to use the bus simultaneously and pass a significant amount of data. In our case, however, we considered that we have two main phases of I2C use: one for reading the MAX32664 sensor and the DS1307 clock, and another for writing to

the screen. Screen writing is always separate from reading, and while the first two operations occur almost simultaneously, they involve reading only a few registers, resulting in very low data volume. This reduces the probability of conflicts to almost zero, leading us to opt for the use of a single I2C bus.

3.3.3 UART

Given the specific requirement of implementing a procedure to read debug data from the board, we chose to use the UART protocol to communicate with the terminal of the ST Link, allowing data transmission and debugging operations.

Among the various UART buses available on the board, we chose UART2 based on past experiences, considering its greater reliability and ease of configuration.

3.3.4 TIM

A key aspect of our device is the timers since we have a strong need to measure time. This will allow us to precisely control the interval between measurements and smoothly execute animations that regulate the sound of the buzzer or the fading of the LED. Additionally, timers play a crucial role in using these two actuators, as we can utilize the PWM generator to generate a varying voltage, thereby adjusting the brightness intensity or the frequency of the emitted sound.

3.3.5 ADC

Lastly, to handle the KY028 temperature sensor, we want to utilize analog technology, so we will use the ADC to measure the voltage values from the thermistor. We will interpret these values in software based on the indications from the datasheet.

3.3.6 PINOUT

Finally, to simplify the hardware design phase, we created a summary of the used pins using the PINOUT visual display provided by the IDE's graphical interface.

We also renamed the pins using User Labels to make their future connections more explicit, except for the pins that will not be used in the circuit, such as those of the UART interface.

- `PA0: ADC1_IN0 -> Temperature`
- `PA2: USART2_TX`
- `PA3: USART2_RX`

- PA8: TIM1_CH1 -> RelaxedLed
- PA9: TIM1_CH2 -> AlertBuzzer
- PB6: I2C1_SCL -> SCL
- PB7: I2C1_SDA -> SDA
- PC0: GPIO_Output -> RST_MAX32664 -> RST
- PC1: GPIO_Output -> MFIO_MAX32664 -> MFIO

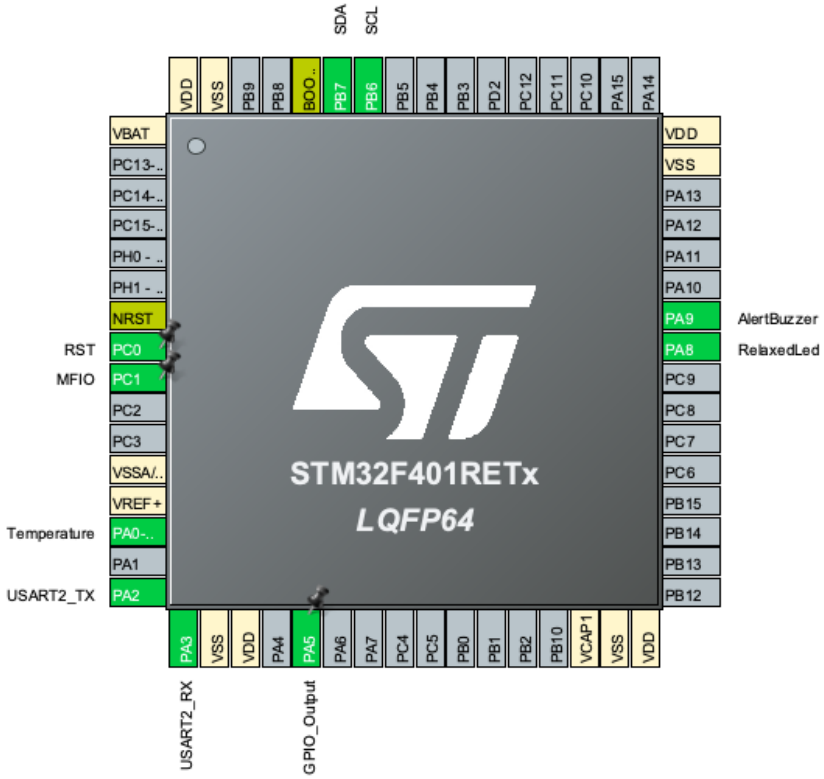

Figure 3.9: PINOUT

Chapter 4

Hardware Architecture

During the hardware design phase, one of the most important tasks was prototyping the circuit using the Fritzing software. This allowed us to create a compact circuit that could fit into a small enclosure.

The first circuit prototype was built using a breadboard, which is a very useful tool for circuit prototyping. It allows for easy and stable electrical connections without the need for soldering, enabling us to easily change connections and test different configurations.

We started by searching online for models of our sensors since they were not natively available in the Fritzing environment, except for the NUCLEO board and the external clock.

We created a new Fritzing project, imported the sensor models, and then built the circuit. Along with this documentation and the project source code, we have also included the Fritzing file and the sensor models.

Within the project, we imported the following components:

- Breadboard
- NUCLEO board
- Temperature sensor
- Heart rate and oxygenation sensor
- External clock
- OLED screen
- LED

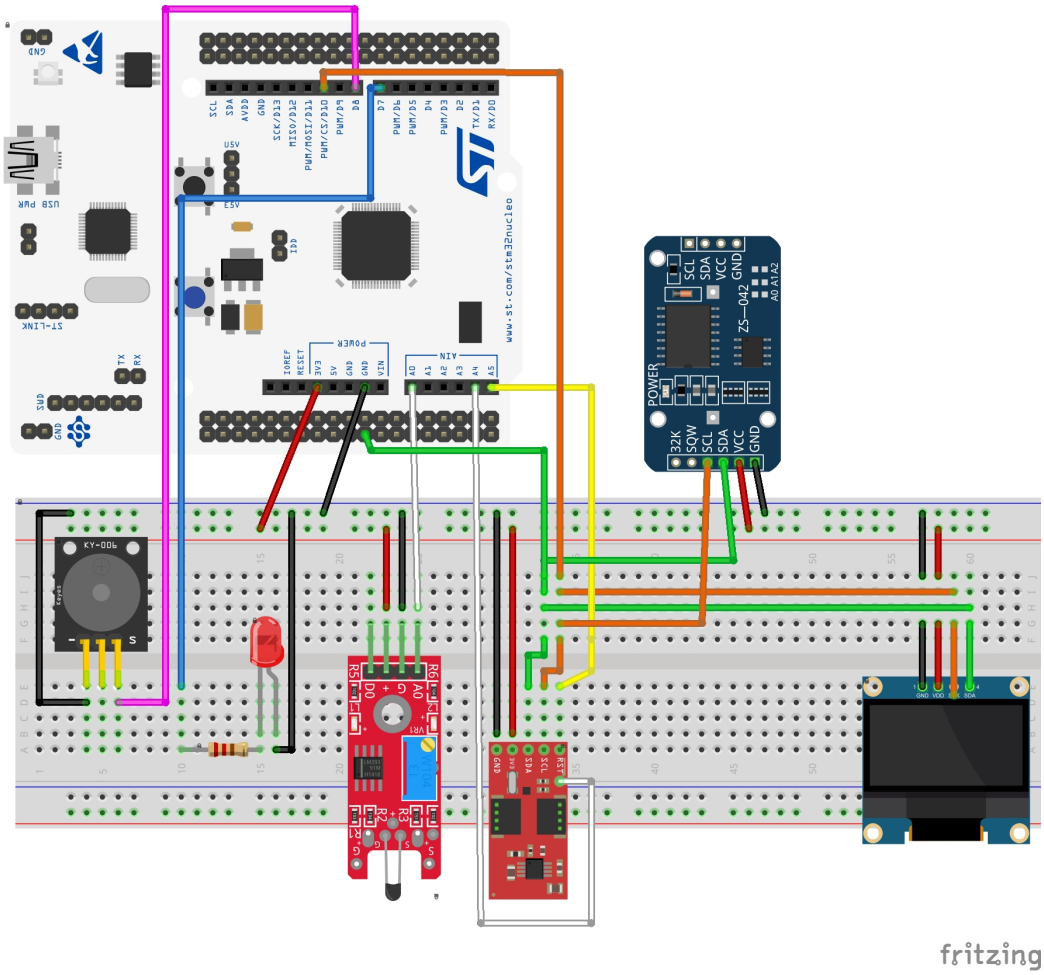

Figure 4.1: Circuit in Fritzing

- 220-ohm resistor
- Passive buzzer

We proceeded to build the circuit, powering everything with 3.3 volts from the board and connecting the ground. We chose this voltage because all our devices operate at this voltage.

Following the PINOUT diagram mentioned above, we identified the pins involved and made the necessary connections.

As shown in Figure 4.1, we tried to keep everything as organized as possible to clearly identify the cable routing, always using right angles whenever possible.

We established conventions to distinguish the cables as follows:

- Red: represents the positive +3.3V
- Black: represents the negative - GND
- Orange: represents the I2C SCL connection
- Green: represents the I2C SDA connection
- White: represents the ADC input for the temperature sensor and the MFIO for the heart rate and oxygenation sensor
- Yellow: represents the RST connection
- Blue: represents TIM channel 1 with PWM
- Pink: represents TIM channel 2 with PWM

After building and verifying the circuit, ensuring that all connections were correct, we considered the number of soldering points required to create the circuit without the breadboard. We concluded that using four terminals for the Red, Black, Orange, and Green wires (positive, negative, SCL, and SDA) would be sufficient. Then, we would make a few soldering points for the LED and the buzzer connections. This would result in a more compact circuit.

An important factor in the circuit design is the length of the white wire connecting the output of the temperature sensor. Since the sensor works in analog mode using ADC, it is important to minimize any voltage drop that can occur over a long copper wire. This drop could alter the measured voltage value.

We inserted a 220-ohm resistor in series with the LED since our LED operates at around 1.8V (Red) or 3V (Blue), while our board provides an output voltage of 5V. Therefore, we add

a resistor in series to cause a voltage drop. The value of this resistor can be calculated using the formula (4.1):

$$R = \frac{V_{pin} - V_{led}}{I} = \frac{3.3V - 1.8V}{0.02A} = 75\Omega \tag{4.1}$$

We could use a 75-ohm resistor, but for simplicity and availability of resistors, we use a 220-ohm resistor.

Another important consideration is calibrating the variable resistor using a small potentiometer integrated into the temperature sensor itself. The precision of the sensor changes based on the value set through this potentiometer.

It should be calibrated once the project is completed using a reliable temperature sensor.

Regarding the hardware, it is necessary to arrange the configuration in such a way that the patient can easily position the temperature sensor and the heart rate and oxygenation sensor in the designated areas mentioned in the "Solution" chapter, without facing any difficulties. The screen should be positioned in a visible location for the user.

A CR2032 lithium battery is placed inside the external clock to keep it powered. After an initial setup, it will maintain the date without any loss until the battery is exhausted, which is estimated to be after several years. This ensures uninterrupted usage of the device for its intended purpose.

Chapter 5

Software Architecture

For our Software Architecture, we have made a series of considerations. First of all, we felt the need to use the singleton pattern to access essential data such as time counters, flag and for some sensors, avoid the use of global variable and in principle in this way we can initialize once the variable and use ever the same.

The source files with their respective headers have been implemented.

- action.c
- ds1307rtc.c
- KY028.c
- main.c
- MAX32664.c
- ssd1306_font.c
- ssd1306.c

Here is the UML diagram for the action.h and main.h files, which show our general implementation. Justifications for the implementation are provided.

action.h

const uint8_t *heartPic
const uint8_t *tempPic
const uint8_t *oxygenPic
const uint8_t *alertPic
enum Measure{
TEMPERATURE_MODE = 0,
HEART_RATE_MODE ,
OXYGEN_MODE
}

computeAverageTemp() : void
computeAverageHeartRate() : void
computeAverageOxygen() : void
showMeasures() : void
showReplaceFinger() : void
compositeString(char*, uint8_t) : void
showTerminalTime(DateTime) : void
showDangerHypoxemia() : void
showDangerTachycardia() : void
showDangerArrhythmia() : void
showDangerFever() : void
showDangerCovid() : void
showDangerHighFever() : void
showDangerHighestFever() : void
showWhichAction() : void
showWhichActionTachycardia() : void
showWhichActionArrhythmia() : void
showWhichActionCovid() : void
showWhichActionFever() : void
showWhichActionHighFever() : void
showWhichActionHighestFever() : void
showWhichActionHypoxemia() : void
showActionTachycardia() : void
showActionHypoxemia() : void
showActionArrhythmia() : void
showActionCovid() : void
showActionFever() : void
showActionHighFever() : void
showActionHighestFever() : void
showInhalation() : void
showExhalation() : void

main.h (first part)

```
enum Threshould{
goodMeasureTime = 10
goodNumberMeasure = 7
countFail = 5
highTemperature = 37
highHeartRate = 60
lowOxygen = 96
lowConfTemp = 20
highConfTemp = 45
lowConfHeartRate = 50
highConfHeartRate = 220
lowConfOxygen = 95
highConfOxygen = 101
acceptableConfidenceHeartRate = 95
}

enum animationDuration{
showDangerDuration = 6000
timeInhalationAndExhalation = 5000
repetitionBreathing = 5
repetitionDuration = 5000
}

typedef struct{
highTemperatureFlag : bool
highHeartRateFlag : bool
lowOxygenFlag : bool
dangerShowing : bool
inhalation : bool
exhalation : bool
}Flag

typedef struct{
tachycardia : bool
hypoxemia : bool
arrhythmia : bool
covid : bool
fever : bool
highFever : bool
highestFever : bool
}Disease
```

main.h (second part)
typedef struct{ countTot : uint32_t goodTemp : uint32_t goodHeartRate : uint32_t goodOxygen : uint32_t countDangerShowing : uint32_t ledCounter : uint32_t repetition : uint32_t badValueTemp : uint32_t badValueHeartRate : uint32_t badValueOxygen : uint32_t sumTemp : float sumHeartRate : uint32_t sumOxygen : uint32_t averageValueTemp : float averageValueHeartRate : uint32_t averageValueOxygen : uint32_t }Measure
getFlag() : Flag* getMeasure() : Measure* getDisease() : Disease* FLAG_Init() : void MEASURE_Init() : void DISEASE_Init() : void resetValue() : void resetAvgValue() : void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef) : void

5.1 action.h

Inside the action.h file, we can first find the definition of some constants that represent bitmap icons of a thermometer, a heart, and a blood drop.

Additionally, we can find an enum that is useful for indicating the current mode:

- Temperature_Mode
- Heart_Rate_Mode
- Oxygen_Mode

Furthermore, within this file, or more precisely in its corresponding source code, we find all the utility functions of our software. This approach has made the writing of the software very convenient as we maximized code reuse and compartmentalization.

We can find functions for calculating average values, functions for displaying values, and functions that also handle displaying alert situations. There are also functions for displaying predefined screens, ensuring clarity.

5.2 main.h

Firstly, we can find the enums **Threshold** and **animationDuration**, which have various constants defined within them based on literature research or design choices. Some of the most important constants include **goodMeasureTime, countFail, low&highConfidenceValue**.

As you can see, a series of data structures have been created to implement the singleton pattern, namely the structs **Flag, Disease, Measure**. Along with their access functions, they enable the storage of various parameters within functions external to the main function.

5.2.1 Flag

These are boolean variables that are changed during measurements to trigger or disable processes within other control flow functions.

5.2.2 Disease

These are boolean variables that indicate the detection of a specific disease by the system. The system is designed to set the conditions to false when subsequent measurements indicate the disappearance of one or more symptoms.

5.2.3 Measure

In this structure, we can find counters and variables that keep track of the system's state. We store summations of values needed for averaging, counters for correct measurements for each sensor, counters for elapsed seconds or milliseconds, and a variable that holds the average of measured values in the past moments.

Chapter 6

Software Interfaces

For the development of the entire project, the main software interface we used was HAL.

The HAL (Hardware Abstraction Layer) interface is a software library provided by STMicroelectronics to simplify application development on STM32 microcontrollers. It provides an abstraction of the microcontroller's specific hardware, allowing developers to write code that is independent of the specific microcontroller being used.

The HAL interface is designed to offer a consistent and portable API for accessing microcontroller peripherals such as GPIO, timers, UART, SPI, I2C, etc. By using the HAL interface, developers can write code that is more readable and portable, reducing the dependency on the specific microcontroller being used.

The key features and advantages of the HAL interface include:

- **Hardware abstraction**: The HAL interface hides the specific details of the microcontroller's hardware, providing a set of generic functions for accessing peripherals. This allows developers to write code that is independent of the specific microcontroller being used and simplifies code portability across different STM32 platforms.
- **Ease of use**: The HAL interface provides well-documented functions and macros for accessing microcontroller peripherals. Developers can use these functions to configure and control peripherals without dealing with complex hardware details.
- **Scalability**: The HAL interface supports various STM32 microcon-

trollers, enabling developers to write code that can be easily adapted for different platforms. This makes it easier to upgrade the microcontroller or migrate the code to a different STM32 platform.

- **Efficiency**: Despite the hardware abstraction, the HAL interface aims to provide optimized performance by minimizing the overhead introduced by the abstraction itself.

6.1 UART2 (Debug)

The UART2 interface is used for debug connection, allowing serial communication between the NUCLEO-F401RE board and a serial terminal on the computer. The UART2 interface enables transmission and reception of data through the TX and RX pins.

To use the UART2 interface, I configured the serial port using the HAL interface. We set the baud rate to 9600 bits/s, word length to 8 bits (including parity), parity to none, stop bit to 1, and enabled global interrupts.

Subsequently, we used the HAL interface functions to send data via UART2. We used the "HAL_UART_Transmit()" function to send data to the serial terminal.

The use of the HAL interface greatly simplifies the management of UART2, providing clear and documented functions to configure and use this peripheral. Furthermore, the abstraction provided by the HAL interface allows for greater code portability, facilitating the migration of the project to other STM32 platforms if needed.

6.2 TIM1 for PWM

Timer TIM1 is used to generate pulse width modulation (PWM) signals. These signals are used to control devices such as DC motors, LEDs, or other components that require precise power or intensity adjustment.

To use TIM1 with channels 1 and 2 for PWM generation, we configured the timer and its channels using the HAL interface. I set the prescaler to 15, the period to 999, and the default duty cycle ratio.

Subsequently, I used the HAL interface functions to enable and disable channels and set the duty cycle ratio. For example, I used the "HAL_TIM_PWM_Start()" function to start the PWM signal generation and the "HAL_TIM_PWM_Stop()"

function to stop it. We also used the "HAL_TIM_SET_COMPARE()" function to set the duty cycle, which in our case varies over time to create LED fading animations or play a musical note.

The use of

the HAL interface simplifies the configuration and control of TIM1 for PWM generation, providing well-defined and documented functions to handle the settings and operations associated with the timer.

6.3 TIM10 for generating interrupts every second

Timer TIM10 is used to generate interrupts at regular intervals, in this case, every second, to handle timed events in your project.

To use TIM10 for generating interrupts every second, we configured the timer using the HAL interface. I set the timer period to generate an interrupt every second, with a prescaler of 1599 and a period of 9999.

Subsequently, I used the HAL interface functions to enable interrupts and handle the callbacks associated with each timed event. For example, I used the "HAL_TIM_Base_Start_IT()" function to start the timer, and implemented the callback function "HAL_TIM_PeriodElapsedCallback()" to handle the event that occurs every second, which in our case involves taking a measurement and any subsequent actions.

The use of the HAL interface simplifies the management of timers and interrupts, providing predefined functions and callbacks to configure and handle timed events.

6.4 TIM11 for generating interrupts every millisecond

Timer TIM11 is used to generate interrupts at regular intervals, such as every millisecond, to handle animations or quick actions in your project.

To use TIM11 for generating interrupts every millisecond, I configured the timer using the HAL interface. I set the timer period to generate an interrupt every millisecond, with a prescaler of 1 and a period of 7999.

Subsequently, I used the HAL interface functions to enable interrupts and handle callbacks, just like with TIM10, but in this case, we handle the event every millisecond. This corresponds to animations, particularly for managing the PWM duty cycle to vary every millisecond or multiples of it controlled by a specific counter, allowing for gradual LED fading and breathing effects, as well as varying the frequency of the buzzer sound emission.

The use of the HAL interface simplifies the management of timers and interrupts, providing functions and callbacks for configuring and handling timed events.

6.5 ADC for reading a temperature sensor

The ADC (Analog-to-Digital Converter) is used to convert an analog signal from a temperature sensor into a digital value that can be read and used by the microcontroller.

To use the ADC for reading a temperature sensor, I configured the ADC using the HAL interface. I set the resolution to 12 bits, the conversion mode to "Continuos Conversion Mode," and other necessary configuration parameters to obtain an accurate reading from the temperature sensor.

Subsequently, I used the HAL interface functions to start the ADC conversion and read the converted value. For example, I used the "HAL_ADC_Start()" function to start the conversion and the "HAL_ADC_GetValue()" function to obtain the converted value.

The use of the HAL interface simplifies the management of the ADC, providing documented functions to configure, start, and read converted values.

6.6 I2C

The I2C (Inter-Integrated Circuit) interface is used to communicate with external devices such as sensors, displays, and other components that support communication via the I2C protocol.

To use the I2C interface to control the heart rate and oxygen sensor, the OLED display, and the external clock, I configured the I2C using the HAL interface. I set the device address, transmission speed, and other configuration parameters necessary to communicate correctly with each device.

Subsequently, I used the HAL interface functions to send and receive data

via I2C. For example, I used the "HAL_I2C_Master_Transmit()" function to send data to the I2C devices and the "HAL_I2C_Master_Receive()" function to receive

data from them.

The use of the HAL interface simplifies the management of I2C communication, providing well-defined functions to configure, send, and receive data from external devices.

Chapter 7

Software Protocols (designed by us)

In order to make our code functional, as mentioned in Chapter 5, we encapsulated the code into several functions, mostly written inside the callbacks of the interrupts we are interested in. Since the device performs operations at regular time intervals, working in this way was crucial.

The main function we implemented is as follows:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *
   ↪ htim){}
```

Inside this function, the main flow control is handled through two **IF** conditions, which check the current timer instance.

```
if (htim->Instance == TIM10){}
if (htim->Instance == TIM11){}
```

7.1 Initialization

Our software first carries out an initialization phase for the device, where all the devices are turned on and configured with initial parameters.

During this phase, we also call specific initialization functions that initialize all the variables created using the singleton pattern.

```
HAL_Init();
SystemClock_Config();
```

```
/*Initialize all configured peripherals*/
MX_GPIO_Init();
MX_I2C1_Init();
MX_TIM1_Init();
MX_USART2_UART_Init();
MX_ADC1_Init();
MX_TIM10_Init();
MX_TIM11_Init();


/*Initialize TIM 10 and 11*/
HAL_TIM_Base_Start_IT(&htim10);
HAL_TIM_Base_Start_IT(&htim11);

/*Initialize ADC*/
HAL_ADC_Start(&hadc1);

/*Initialize Oled_SSD1306*/
ssd1306_Init();

/*Initialize MAX32664*/
MAX32664* max = getSensor();

/*Initialize Measure, Flag and Disease Structures*/
FLAG_Init();
MEASURE_Init();
DISEASE_Init();

/*Initialize and Start DS1307RTC*/
ds1307rtc_init();
ds1307rtc_start();

/*Define the ADC conversion in PollingMode*/
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
```

After the initialization phase, the only static action that is performed is to call the function **"showMeasures()"** once, which is responsible for displaying the measurements on the screen. In the case of the first execution where no data is available, it displays dashes for all the parameters.

7.2 TIM 10

To handle this case, we are inside the corresponding **IF** statement. The first thing to consider is that all the operations we are about to see are executed once every second, which is the interval we have chosen between one measurement and another.

7.2.1 Measurement

After calling the variables of the singleton pattern, we proceed with the measurements by retrieving data from the appropriate sensors.

```
measure->countTot += 1;

/*Put the value of temperature in this variable, from
   ↪ the function 'getTemperature'*/
temperature = getTemperature();
/*I check that the value of temperature is between a min
   ↪  value and a max value*/
/*to don't take into account the out-of-bounds value for
   ↪  the mean.*/
if ((temperature< highConfTemp) && (temperature >
   ↪ lowConfTemp)){
        measure->sumTemp += temperature;
        measure->goodTemp++;
}

/*Read value of MAX32664*/
read_sensor(max);

uint32_t heartRate = max->heart_rate;
uint32_t oxygen = max->oxygen;
uint32_t confidence_heart_rate = max->
   ↪ confidence_heart_rate;

/*Check that the value of heart rate is acceptable*/
if ((heartRate < highConfHeartRate) && (heartRate >
   ↪ lowConfHeartRate) && (confidence_heart_rate >
   ↪ acceptableConfidenceHeartRate)){
        measure->sumHeartRate += heartRate;
        measure->goodHeartRate++;
}
/*Check that the value of oxygen is acceptable*/
```

```
if ((oxygen < highConfOxygen) && (oxygen > lowConfOxygen
   ↪ )){
        measure->sumOxygen += oxygen;
        measure->goodOxygen++;
}

sprintf(valueTmp, "Temperatura: %.2f\n\r", temperature);
HAL_UART_Transmit(&huart2,valueTmp , strlen(valueTmp),
   ↪ HAL_MAX_DELAY);

sprintf(valueHeart, "HeartRate: %d\n\r", heartRate);
HAL_UART_Transmit(&huart2,valueHeart , strlen(valueHeart
   ↪ ), HAL_MAX_DELAY);

sprintf(valueOxygen, "Oxygen: %d\n\r", oxygen);
HAL_UART_Transmit(&huart2,valueOxygen , strlen(
   ↪ valueOxygen), HAL_MAX_DELAY);

sprintf(valueTmp, "COUNT: %d\r\n", measure->countTot);
HAL_UART_Transmit(&huart2, valueTmp, strlen(valueTmp),
   ↪ HAL_MAX_DELAY);
```

In the last lines, UART_Transmit commands are used to print the measurements on the terminal. While the OLED screen updates the measurements every 10 seconds, the ones from the USB need to be transmitted all at once.

It is worth noting that the first instruction is increment a counter that will allow us to create flow controls, for example, every x seconds.

7.2.2 10 Measurements made

```
if (measure->countTot >= goodMeasureTime){
        char str[] = "Nuova misura disponibile\r\n";
        HAL_UART_Transmit(&huart2, str, strlen(str),
           ↪ HAL_MAX_DELAY);
        /*Check that the measures of temperature are
           ↪ enough*/
        if (measure->goodTemp >= goodNumberMeasure){
                measure->badValueTemp = 0;
                previousValueTemp = measure->
                   ↪ averageValueTemp;
                computeAverageTemp();
```

```
                /*If my average value is over the
                   ↪ threshold, set the flag = true*/
                flags->highTemperatureFlag = (measure->
                   ↪ averageValueTemp < highTemperature
                   ↪ ) ? false : true;
        }else{
                /*If the measures is not enough,
                   ↪ increment the counter for error
                   ↪ measure and reset the average
                   ↪ value*/
                measure->badValueTemp++;
                measure->averageValueTemp = 0;      // In
                   ↪  this way on my screen I will show
                   ↪  that the measure is wrong
                flags->highTemperatureFlag = false;
        }
        /*Check that the measures of heart rate are
           ↪ enough*/
        if (measure->goodHeartRate >= goodNumberMeasure)
           ↪ {
                measure->badValueHeartRate = 0;
                previousValueHeartRate = measure->
                   ↪ averageValueHeartRate;
                computeAverageHeartRate();
                /*If my average value is over the
                   ↪ threshold, set the flag = true*/
                flags->highHeartRateFlag = (measure->
                   ↪ averageValueHeartRate <
                   ↪ highHeartRate) ? false : true;
        }else{
                /*If the measures is not enough,
                   ↪ increment the counter for error
                   ↪ measure and reset the average
                   ↪ value*/
                measure->badValueHeartRate++;
                measure->averageValueHeartRate = 0;
                /*In this way on my screen I will show
                   ↪ that the measure is wrong*/
                flags->highHeartRateFlag = false;
        }
        /*Check that the measures of oxygen are enough*/
        if (measure->goodOxygen >= goodNumberMeasure){
                measure->badValueOxygen = 0;
```

```
                previousValueOxygen = measure->
                    ↪ averageValueOxygen;
                computeAverageOxygen();
                /*If my average value is over the
                    ↪ threshold, set the flag = true*/
                flags->lowOxygenFlag = (measure->
                    ↪ averageValueOxygen < lowOxygen) ?
                    ↪ true : false;
        }else{
                /*If the measures is not enough,
                    ↪ increment the counter for error
                    ↪ measure and reset the average
                    ↪ value*/
                measure->badValueOxygen++;
                measure->averageValueOxygen = 0;
                /*In this way on my screen I will show
                    ↪ that the measure is wrong*/
                flags->lowOxygenFlag = false;
        }
        /*This line is important because for each
            ↪ measurement we initialize the value to
            ↪ compute the averages*/
        resetValue();
}
```

Analyzing the condition preceding the code snippet, the threshold of **goodMeasureTime** is set to **10**. Generally, in our code, we do not have fixed variables but rather declare them as enumerables.

Therefore, every 10 seconds, we check how many valid data we have recorded by using the corresponding variables **goodTemp**, **goodHeartRate**, and **goodOxygen**, which are compared with **goodMeasureNumber** set to **6**. If a measurement is considered valid, we calculate the average measurement using the **computeAverege____()** functions and determine whether this value should enable a flag for triggering an emergency call.

7.2.3 Exception handling

There are two cases of exceptions to handle: one is when the finger is not detected, and the other is when warning messages need to be shown to the user due to emerging issues.

To distinguish between these two conditions, we use the **IF** statement:

```
if (((measure->badValueHeartRate >= countFail) || (
   ↪ measure->badValueOxygen >= countFail) || (measure
   ↪ ->badValueTemp >= countFail)) && (measure->
   ↪ countTot >= goodMeasureTime)){

/*Finger not detect Case*/

}else if(measure->countTot >= goodMeasureTime){

/*Alert case*/

}
```

7.2.4 Finger not detect

In this case, we display a message asking the user to reposition their finger, reset the calculated values, and set the counter to 0 to discard all measurements that did not correctly detect the finger.

```
showReplaceFinger();
/*This line is important because I must reset the avg
   ↪ value, otherwise I would still have these value*/
resetAvgValue();
/*This line is important because I must set to 0 the
   ↪ counter else if I'm in this state*/
measure->countTot = 0;
/*I will have a loop because compute always the avg
   ↪ values but I have only one value.*/
```

7.2.5 Raise some alert

In this part of our code, we evaluate the status of the activated flags from earlier operations. We perform visualization operations for the corresponding alerts and activate TIM 11 for animation management. It is important to deactivate TIM 10 while handling emergencies to pause the measurement detection.

To avoid redundancy, only the handling of one exception is mentioned here.

```
/*If the values are good I can show the averages value
   ↪ and I must check which flags are active*/
measure->countTot = 0;
```

```
/*0-0-0*/
if (!flags->highTemperatureFlag && !flags->
   ↪ highHeartRateFlag && !flags->lowOxygenFlag){
    /*TUTTO BENE*/
    showTerminalTime(datetime);
    showMeasures();
}
/*0-0-1*/
if (!flags->highTemperatureFlag && !flags->
   ↪ highHeartRateFlag && flags->lowOxygenFlag){
    /*IPOSSIEMIA*/
    /*check that isn't a first time*/
    if(previousValueOxygen < lowOxygen){ // if isn't a
       ↪ first time, I don't show the danger screen
        showTerminalTime(datetime);
        showMeasures();
    }else{
        /*show the danger screen (IPOSSIEMIA)*/
        flags->dangerShowing = true;
        diseases->hypoxemia = true;
        showDangerHypoxemia();
        HAL_TIM_Base_Stop_IT(&htim10);
        HAL_TIM_Base_Start_IT(&htim11);
    }
}
/*0-1-0*/
if (!flags->highTemperatureFlag && flags->
   ↪ highHeartRateFlag && !flags->lowOxygenFlag){}
/*0-1-1*/
if (!flags->highTemperatureFlag && flags->
   ↪ highHeartRateFlag && flags->lowOxygenFlag){}
/*1-0-0*/
if (flags->highTemperatureFlag && !flags->
   ↪ highHeartRateFlag && !flags->lowOxygenFlag){}
/*1-0-1*/
if (flags->highTemperatureFlag && !flags->
   ↪ highHeartRateFlag && flags->lowOxygenFlag){}
/*1-1-0*/
if (flags->highTemperatureFlag && flags->
   ↪ highHeartRateFlag && !flags->lowOxygenFlag){}
/*1-1-1*/
if (flags->highTemperatureFlag && flags->
   ↪ highHeartRateFlag && flags->lowOxygenFlag){}
```

7.3 TIM 11

This timer is dedicated to animation management. When invoked, it counts milliseconds, allowing us to create smooth animations controlled millisecond by millisecond.

After calling the instances of the singleton pattern, I check the flags and activate animations conditionally by enabling the channels of TIM 1, depending on whether we need to manage the buzzer (Channel 2) or the LED (Channel 1). The animations are regulated by timing based on the counter of TIM 11.

```
HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_2);

if (flags->dangerShowing){
        if (measures->countDangerShowing <=
           ↪ showDangerDuration){
                if (measures->countDangerShowing == (
                   ↪ showDangerDuration/2))
                        showWhichAction();

                __HAL_TIM_SET_COMPARE(&htim1,
                   ↪ TIM_CHANNEL_2, 2500);
                measures->countDangerShowing++;
        HAL_TIM_OC_Stop(&htim1, TIM_CHANNEL_2);
                HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_1);

        }else{
                flags->dangerShowing = false;
                measures->countDangerShowing = 0;
                __HAL_TIM_SET_COMPARE(&htim1,
                   ↪ TIM_CHANNEL_2, measures->
                   ↪ countDangerShowing);
        }
}else if(diseases->arrhythmia && !flags->dangerShowing
   ↪ && (measures->repetition <= repetitionDuration)){
        /*mostra azione per aritmia*/
        if (measures->repetition == 0)
                showActionArrhythmia();
        measures->repetition++;
}else if(diseases->covid && !flags->dangerShowing && (
   ↪ measures->repetition <= repetitionDuration)){
}else if(diseases->fever && !flags->dangerShowing && (
   ↪ measures->repetition <= repetitionDuration)){
}else if(diseases->highFever && !flags->dangerShowing &&
```

```
   ↪  (measures->repetition <= repetitionDuration)){
}else if(diseases->highestFever && !flags->dangerShowing
   ↪  && (measures->repetition <= repetitionDuration)){
}else if(diseases->hypoxemia && !flags->dangerShowing &&
   ↪  (measures->repetition <= repetitionDuration)){
}else if (diseases->tachycardia && !flags->dangerShowing
   ↪  && (measures->repetition <= repetitionBreathing))
   ↪ {
}else{
        showMeasures();
        measures->repetition = 0;
        DISEASE_Init();
        HAL_TIM_Base_Stop_IT(&htim11);
        HAL_TIM_Base_Start_IT(&htim10);
    HAL_TIM_OC_Stop(&htim1, TIM_CHANNEL_1);
}
```

Chapter 8

Power Dissipation

8.1 What is?

The term "Power dissipation" refers to the conversion of electrical energy into thermal energy in a component, device, or system. When an electric current flows through a component or device, it encounters resistance, which results in the generation of heat. This heat is a byproduct of electrical energy being converted into thermal energy.

Power dissipation occurs due to various factors such as resistive losses, switching losses, leakage currents, and other internal losses within electronic components or systems. These losses can be caused by intrinsic resistance in conductors, imperfect components, or inefficiencies in energy conversion.

Power dissipation is typically measured in Watts (W) and is calculated using Ohm's Law, which states that power (P) is equal to the product of voltage (V) and current (I): $P = V \cdot I$. Alternatively, power dissipation can be calculated as the square of the current (I) multiplied by the resistance: $P = I^2 \cdot R$

Managing power dissipation is crucial in electronic systems as excessive heat buildup can degrade performance, cause premature component failure, or even pose safety hazards. Thermal management techniques such as heat sinks, fans, and proper airflow are employed to dissipate the generated heat and keep components within safe operating temperatures.

Information about power dissipation for sensors is typically provided in the datasheet or technical specifications provided by the sensor manufacturer. When looking for information on power dissipation for sensors, you can refer to the following sections or parameters in the datasheet:

- Electrical characteristics: This section often includes parameters related to power consumption or power dissipation of the sensor. Look for specifications such as "Supply current," "Operating current," or "Power consumption." These values indicate the typical or maximum current or power consumed by the sensor during operation.
- Supply voltage: The datasheet may specify the required supply voltage for the sensor. By multiplying the supply voltage by the consumed current, you can roughly calculate the power dissipation of the sensor.
- Power modes: Some sensors may have different power modes or operating states that consume varying amounts of power. The datasheet may provide details on these modes and their associated power consumption or dissipation values.
- Thermal considerations: Some datasheets may include information on the thermal characteristics of the sensor, such as thermal resistance or thermal dissipation. This information can provide an understanding of how the sensor dissipates heat and indirectly provide information about power dissipation.

8.2 Power Calculation

The actual power used by various components can be calculated using the formula 8.1.

$$P = \frac{V^2}{R} \tag{8.1}$$

Based on this equation, we can make various assumptions that differ for different components. In general, for more complex devices like the I2C devices we are using, the resistance varies depending on the operating condition of the peripheral. Therefore, we need to consider different usage scenarios.

In addition to the peripherals, it is essential to consider the power consumption of the board we are using.

8.2.1 NUCLEO-F401RE

The IDE of the STM32 board allows us to input all the peripherals used in the project and calculate the average current consumption by the board.

Furthermore, all these measurements are displayed in a graph, and for our specific case, the graph is as follows:

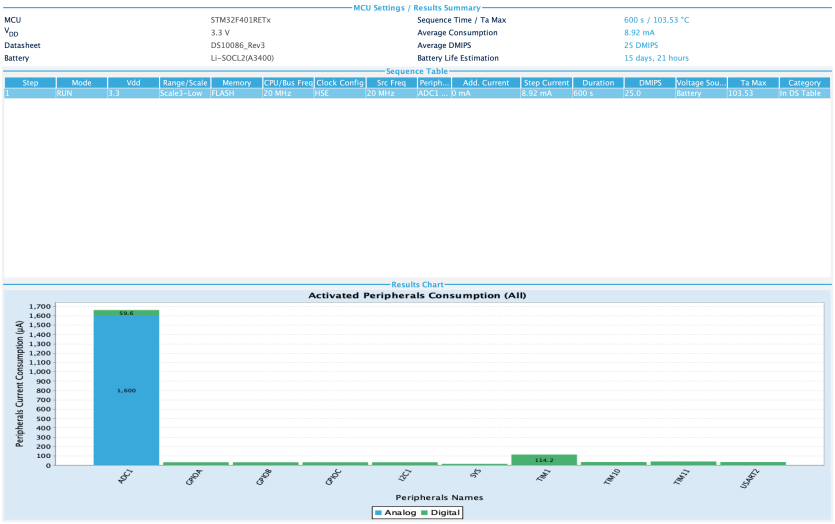

Figure 8.1: Nucleo F04 power consumption

$$P_{NUCLEO-F401RE} = V \cdot I = 3.3 \cdot 0.00892A = 29.43mW \tag{8.2}$$

8.2.2 KY-028

The device in question is purely resistive, consisting of 6 resistors and 2 LEDs.

Its basic operation tells us that one of the resistors (the most important one) varies based on the detected temperature. Therefore, we cannot assume a standardized power consumption under all usage conditions. Instead, we will calculate its consumption at a temperature of approximately 30°C.

We have calculated the resistance value of the circuit between the + and GND terminals, and found it to be approximately 135 KΩ.

$$P_{KY028} = \frac{3.3^2}{135000} = 0.080mW \tag{8.3}$$

8.2.3 MAX32664

In the datasheet of this sensor, under the section titled "Absolute Maximum Ratings," it states: Continuous Package Power Dissipation 24 TQFN-EP on

a multilayer board with a maximum ambient temperature (T_A) of +70°C and a derating factor of 16.3 µW/°C above +70°C is 1305 µW. Therefore, under our conditions, the power dissipation is:

$$P_{MAX32664} = 1305mW \tag{8.4}$$

8.2.4 DS1307

Regarding the DS1307 sensor, the documentation states that the current usage in active mode of the sensor is 1.5 mA. From this, we can calculate its dissipated power:

$$P_{DS1307} = V \cdot I = 3.3V \cdot 0.0015A = 4.95mW \tag{8.5}$$

8.2.5 SSD1306

From the documentation, we can see that with the contrast set to FFh, the average current consumption is 430 µA and the maximum is 780 µA. Considering that the screen is not fully lit in our interfaces, but rather about halfway, we calculate the dissipated power based on the average case:

$$P_{SSD1306} = V \cdot I = 3.3V \cdot 0.000430A = 1.419mW \tag{8.6}$$

8.2.6 LED

The dissipated power for the red LED we are using is:

$$P = V \cdot I = 1.8V \cdot 0.0068A = 0.012W \tag{8.7}$$

However, for our LED, a resistor of 220Ω has been added, which brings the total resistance to 484.70Ω.

$$P_{LED_{Instantaneous}} = \frac{3.3^2}{484.70} = 22.46mW \tag{8.8}$$

Another assumption to consider is that the LED is used in PWM mode, which implies the use of a different input voltage. However, the advantage we can leverage from this condition is that our LED starts with a duty cycle

of 0 and gradually reaches its maximum over a period of 5000 ms. This means that the total energy is simply the area under the curve of the function:

$$P(t) = \frac{V^2 \cdot t}{R \cdot T_{\text{Duration}}} \tag{8.9}$$

where $T_{\text{Duration}} = 5000$ ms in our case, and t represents the time instant of the animation.

Therefore, the integral that will be calculated is the integral of the function $P(t)$ between the limits of 0 and 5000, which represent the start and end values of the animation:

$$E_{\text{LED-Fade}} = \int_0^{5000} P(t)dt = \frac{3.3^2 \cdot t}{R \cdot 5000} = 37.5\text{W} \cdot 5000\text{ms} \tag{8.10}$$

The value of the calculated integral indicates the total energy expended to gradually turn on the LED using a fading animation.

We could also define the average power of the LED as $P_{LED} = \frac{E_{LED}}{5000\text{ms}} = 7.5\text{mW}$.

However, in order to calculate the total energy dissipation, we will not consider the LED among the power consumption values. Instead, we should assume that we need to spend the energy of 7.5 mW for the entire duration of the breathing animation, which is 60 seconds:

$$E_{\text{LED-TOT}} = P_{LED} \cdot \frac{60}{3600} = 0.125\text{mWh} \tag{8.11}$$

8.2.7 Buzzer

After analyzing the documentation for the buzzer, we found that the power consumption of a passive buzzer is 8Ω. Based on this value, we can calculate the dissipated power. Since the buzzer also operates in PWM mode, its power dissipation is:

$$P_{Buzzer} = \frac{0.33^2}{8} = 13.61mW \tag{8.12}$$

The voltage value is 0.33 because the chosen duty cycle is 10

For the energy calculation, the duration is 5000 ms. In terms of watt-hours, it will be:

$$E_{Buzzer} = P_{Buzzer} \cdot \frac{5}{3600} = 0.019mWh \tag{8.13}$$

8.2.8 Final Calculation

For the final calculation, the power dissipation is expressed in watt-hours, which is the sum of all components:

$$P_{\text{TOTAL}} = P_{\text{NUCLEO-F401RE}} + P_{KY028} + P_{MAX32664} + P_{SSD1306} + P_{DS1307} \tag{8.14}$$

$$P_{\text{TOTAL}} = (29.43 + 0.080 + 1305 + 1.419 + 4.95)\text{mW} = 1340\text{mW} \tag{8.15}$$

One final consideration is that, for the energy calculation, all sensors in this sum are continuously active. Therefore, calculating the hourly energy, we have:

$$E_{\text{TOTAL}} = P_{\text{TOTAL}} \cdot 1\text{h} = 1340\text{mWh} \tag{8.16}$$

So, in conclusion, we have the following power consumption values:

- Board and peripherals: 1340 mWh per hour of usage
- LED: 0.125 mWh per cycle of tachycardia management
- Buzzer: 0.019 mWh per alarm signaled

Bibliography

[1] Lai F, Li X, Liu T, Wang X, Wang Q, Chen S, Wei S, Xiong Y, Hou Q, Zeng X, Yang Y, Li Y, Lin Y, Yang X. Soglie di febbre diagnostica ottimali utilizzando termometri a infrarossi senza contatto sotto COVID-19. Fronte Salute Pubblica. 24 novembre 2022;10:985553. doi: 10.3389/f-pubh.2022.985553. PMID: 36504995; PMCID: PMC9730337.

[2] Spodick DH. frequenza cardiaca sinusale normale: soglie di frequenza appropriate per tachicardia sinusale e bradicardia. South Med J. 1996 Jul;89(7):666-7. doi: 10.1097/00007611-199607000-00003. PMID: 8685750.

[3] Goyal D, Inada-Kim M, Mansab F, Iqbal A, McKinstry B, Naasan AP, Millar C, Thomas S, Bhatti S, Lasserson D, Burke D. Migliorare l'identificazione precoce della polmonite da COVID-19: una revisione narrativa. BMJ Open Respir Res. 2021 Nov;8(1):e000911. doi: 10.1136/bmjresp-2021-000911. PMID: 34740942; PMCID: PMC8573292.

[4] SSD1306_datasheet.pdf

[5] MAX32664_datasheet.pdf

[6] KY028_datasheet.pdf

[7] DS1307_datasheet.pdf