Dídac Roda Pitarg

DEVELOPMENT OF A LOW-POWER WEARABLE DEVICE FOR PERSONALIZED POLLUTION EXPOSURE ASSESSMENT

FINAL DEGREE PROJECT

Directed by Dr. Eduard Llobet

Bachelor's Degree in Telecommunication Systems and Services Engineering

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

En primer lloc, m'agradaria donar les gràcies a la meva família, pel seu recolzament, la seva compressió i fe en mi al llarg d'aquests durs mesos.

Agrair al meu tutor, l'Eduard, per la seva orientació i paciència durant aquest trajecte.

També m'agradaria expressar el meu més sincer agraïment a la Noelia per brindar-me l'oportunitat de treballar com a enginyer en el projecte OnBREATHE. Aquesta experiència ha estat i està sent increïblement enriquidora.

Finalment, un reconeixement especial a en Xavi pel seu suport i ànims en els meus esforços. La seva presència ha estat una font de força i motivació, especialment en els moments difícils.

Abstract

Air pollution poses a significant threat to global health, causing millions of premature deaths annually. Traditional air quality monitoring stations provide a general picture, but they fail to capture the variations in an individual's exposure throughout the day. This project addresses this gap by presenting the development and initial testing of a low-power wearable air quality device for personalized monitoring.

The project aligns with the Sustainable Development Goals (SDGs) by promoting good health and well-being, fostering technological innovation, and contributing to sustainable cities and communities. It contributes to the *OnBREATHE* project, which aims to create a wearable tool for tracking air quality and health markers in patients with chronic respiratory illnesses.

This work focusses on developing a wearable microsensor that continuously monitors air quality for personalized exposure assessment. A custom-designed printed circuit board (PCB) incorporates a BME680 sensor to measure temperature, pressure, humidity, and volatile organic compounds (VOCs). The microcontroller transmits data via Bluetooth Low Energy (BLE) to a mobile application (under development by a collaborator).

Initial testing of the PCB prototype delivered promising results. The BME680 sensor successfully collected environmental data, with clear distinctions observed between indoor and outdoor readings. This demonstrates the device's potential to capture variations in air quality throughout the day. However, achieving higher accuracy levels for the Indoor Air Quality (IAQ) sensor calibration remains a challenge, limiting the precision of b-VOC and CO_2 equivalent readings.

Overall, the project successfully developed a functional prototype for a low-power wearable air quality monitor. Future work will focus on refining the calibration process, conducting wearability studies, integrating additional sensors, and validating the device's performance in real-world settings. This innovative wearable device has the potential to empower individuals to manage their exposure to pollutants and contribute to improved air quality for all.

Resumen

La contaminación atmosférica supone una importante amenaza para la salud mundial y causa millones de muertes prematuras al año. Las estaciones tradicionales de control de la calidad del aire ofrecen una imagen general, pero no captan las variaciones de la exposición de un individuo a lo largo del día. Este proyecto aborda esta carencia presentando el desarrollo y las pruebas iniciales de un dispositivo portátil de bajo consumo para la monitorización personalizada de la calidad del aire.

El proyecto se alinea con los Objetivos de Desarrollo Sostenible (ODS) mediante la promoción de la buena salud y el bienestar, el fomento de la innovación tecnológica y la contribución a ciudades y comunidades sostenibles. Contribuye al proyecto *OnBREATHE*, cuyo objetivo es crear una herramienta portátil para el seguimiento de la calidad del aire y los marcadores de salud en pacientes con enfermedades respiratorias crónicas.

Este trabajo se centra en el desarrollo de un microsensor portátil que monitoriza continuamente la calidad del aire para una evaluación personalizada de la exposición. Una placa de circuito impreso (PCB) diseñada a medida incorpora un sensor BME680 para medir la temperatura, la presión, la humedad y los compuestos orgánicos volátiles (COV). El microcontrolador transmite los datos a través de Bluetooth Low Energy (BLE) a una aplicación móvil (en desarrollo por un colaborador).

Las pruebas iniciales del sensor arrojaron resultados prometedores. El sensor BME680 recogió con éxito datos ambientales, observándose claras diferencias entre las lecturas en interiores y exteriores. Esto demuestra el potencial del dispositivo para captar las variaciones de la calidad del aire a lo largo del día. Sin embargo, sigue siendo un reto lograr mayores niveles de precisión en la calibración del sensor de calidad del aire interior (IAQ), lo que limita la precisión de las lecturas de b-VOC y CO_2 equivalente.

En resumen, se ha desarrollado con éxito un prototipo funcional de monitor portátil de calidad del aire de bajo consumo. El trabajo futuro se centrará en perfeccionar el proceso de calibración, realizar estudios de comodidad, integrar sensores adicionales y validar el rendimiento del dispositivo en entornos reales. Este innovador dispositivo portátil puede ayudar a las personas a controlar su exposición a los contaminantes y contribuir a mejorar la calidad del aire para todos.

Resum

La contaminació atmosfèrica suposa una important amenaça per a la salut mundial i causa milions de morts prematures a l'any. Les estacions tradicionals de control de la qualitat de l'aire ofereixen una imatge general, però no capten les variacions de l'exposició d'un individu al llarg del dia. Aquest projecte aborda aquesta mancança presentant el desenvolupament i les proves inicials d'un dispositiu portàtil de baix consum per al monitoratge personalitzat de la qualitat de l'aire.

El projecte s'alinea amb els Objectius de Desenvolupament Sostenible (ODS) mitjançant la promoció de la bona salut i el benestar, el foment de la innovació tecnològica i la contribució a ciutats i comunitats sostenibles. Contribueix al projecte *OnBREATHE*, l'objectiu del qual és crear una eina portàtil per al seguiment de la qualitat de l'aire i els marcadors de salut en pacients amb malalties respiratòries cròniques.

Aquest treball se centra en el desenvolupament d'un microsensor portàtil que monitora contínuament la qualitat de l'aire per a una avaluació personalitzada de l'exposició. Una placa de circuit imprès (PCB) dissenyada a mesura incorpora un sensor BME680 per a mesurar la temperatura, la pressió, la humitat i els compostos orgànics volàtils (COV). El microcontrolador transmet les dades a través de *Bluetooth Low Energy* (BLE) a una aplicació mòbil (en desenvolupament per un col·laborador).

Les proves inicials del sensor van abocar resultats prometedors. El sensor BME680 va recollir amb èxit dades ambientals, observant-se clares diferències entre les lectures en interiors i exteriors. Això demostra el potencial del dispositiu per a captar les variacions de la qualitat de l'aire al llarg del dia. No obstant això, continua sent un repte aconseguir majors nivells de precisió en el calibratge del sensor de qualitat de l'aire interior (IAQ), la qual cosa limita la precisió de les lectures de b-VOC i CO_2 equivalent.

En resum, s'ha desenvolupat amb èxit un prototip funcional de monitor portàtil de qualitat de l'aire de baix consum. El treball futur se centrarà a perfeccionar el procés de calibratge, realitzar estudis de comoditat, integrar sensors addicionals i validar el rendiment del dispositiu en entorns reals. Aquest innovador dispositiu portàtil pot ajudar les persones a controlar la seva exposició als contaminants i contribuir a millorar la qualitat de l'aire per a tots.

Contents

1 Introduction 1
 1.1 Sustainable Development Goals 1
 1.2 The *OnBREATHE* project 2
 1.3 Motivation 3
 1.4 Goals 4
2 Theoretical framework 5
 2.1 VOCs 5
 2.2 Internet of Things 6
 2.3 Bluetooth Low Energy 7
 2.4 Inter-Integrated Circuit 10
 2.5 Printed Circuit Board 11
  2.5.1 Microcontroller (STM32WB5MMG) 12
  2.5.2 LDO regulator (NCP705MT33) 14
  2.5.3 Gas sensor (BME680) 15
  2.5.4 Battery gauge (LTC2942-1) 16
  2.5.5 Battery charger (MCP73831) 18
3 Software Development 21
 3.1 STM32CubeIDE 21
  3.1.1 Inter-Integrated Circuit 26
  3.1.2 Gas sensor (BME680) 27
  3.1.3 Bluetooth Low Energy 32
  3.1.4 Battery gauge (LTC2942-1) 35
4 PCB Miniaturization 37
 4.1 Eagle 37
5 Validation and results 39
6 Conclusions and future perspectives 41
Budget 43
References 45
Appendices 47
 Appendix I – PCB 47
 Appendix II – STM32 code 51
 Appendix III – BLE App Inventor Android app 73

List of Abbreviations

ATT	Attribute Protocol
BLE	Bluetooth Low Energy
BSEC	Bosch Software Environmental Cluster
GATT	Generic ATTribute Profile
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
IAQ	Index for Air Quality
IC	Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
LDO	Low-dropout
MCU	Microcontroller Unit
MOX	Metal Oxide
PCB	Printed Circuit Board
SCL	Serial Clock
SDA	Serial Data
SDG	Sustainable Development Goal
UUID	Universally Unique Identifier
VOC	Volatile Organic Compound

List of Figures

Figure 1. SDGs related to the project. 1
Figure 2. Graphical abstract of the OnBREATHE project. 3
Figure 3. Main sources of VOCs. 6
Figure 4. Environments of IoT applications (adopted from [8]). 6
Figure 5. Illustrative example of BLE advertisement. 8
Figure 6. Generic Attribute Profile hierarchy. 9
Figure 7. I2C block diagram [11]. 10
Figure 8. I2C START signal [11]. 10
Figure 9. I2C STOP signal [11]. 11
Figure 10. I2C byte acknowledgement signal [11]. 11
Figure 11. I2C packet. 11
Figure 12. High-level overview of the PCB. 12
Figure 13. Block diagram of the STM32WB5MMG module [12]. 13
Figure 14. NCP705 Typical Application circuit [14]. 14
Figure 15. BSEC-Derived Air Quality Index classification, impact, and suggested actions [15]. 16
Figure 16. LTC2942-1 typical application circuit [16]. 17
Figure 17. LTC2942-1's register map and control register [16]. 17
Figure 18. MCP73831 typical application circuit [17]. 18
Figure 19. MCP73831 operating state diagram (adopted from [17]). 19
Figure 20. STM32CubeIDE welcome page. 22
Figure 21. STM32CubeIDE target selection screen. 22
Figure 22. STM32CubeIDE project creation last step. 23
Figure 23. STM32CubeIDE Development Configuration Tool screen. 23
Figure 24. Close-up of the STLINK-V3SET highlighting the various communication interfaces it supports [18]. 24
Figure 25. Connection of the STLINK-V3SET programmer to the PCB using the clip clamp. 25
Figure 26. STLinkUpgrade screen. 25
Figure 27. STM32CubeIDE I2C1 GPIO settings. 26
Figure 28. STM32CubeIDE toolbar. 26
Figure 29. STM32CubeIDE confirmation prompt for generating code. 26
Figure 30. STM32CubeIDE project with the BME680 library. 27
Figure 31. STM32CubeIDE project settings view. 28
Figure 32. Errors when using hardware floating-point operations. 30
Figure 33. STM32CubeIDE project MCU Settings. 31
Figure 34. BSEC data output. 31
Figure 35. STM32CubeIDE STM32_WPAN package configuration. 32
Figure 36. STM32CubeIDE task configuration. 33
Figure 37. a) FUS information before wireless stack installation. b) FUS information after wireless stack installation. 34
Figure 38. Reading BLE characteristics from the wearable with a commercial app. 35
Figure 39. Board view of the miniaturized PCB. 37
Figure 40. PCB environmental measurements after BME680's calibration. 40

List of Tables

Table 1. Types of IoT networks. 7
Table 2. Differences between Bluetooth Classic and Bluetooth Low Energy (adopted from [10]). 8
Table 3. Characteristics of the components used in the PCB. 12
Table 4. Low-power mode comparison for the STM32WB5MMG (adopted from [13]). 14
Table 5. Comparison of BME680 and BSEC output parameters. 15
Table 6. GATT services and characteristics for data transmission. 33
Table 7. PCB production costs. 44

List of Codes

Code 1. BME680 read and write functions via I2C. 29
Code 2. BME680 and BSEC measurements reading. 30
Code 3. BLE characteristic update. 34
Code 4. Battery gauge configuration lines. 36
Code 5. Battery gauge battery percentage functions. 36

1 Introduction

The term “air pollution” refers to the contamination of the atmosphere by chemical, biological, or physical agents [1]. Pollution exposure in Europe led to 238,000 premature deaths in 2021, according to the World Health Organization (WHO). The situation wasn’t much better in 2019, where 99% of the global population was living in places with air quality levels above the WHO standards [2].

Aside from early demise, air pollution also leads to morbidity. People suffer from illnesses brought on by exposure to contaminants, placing a heavy financial strain on health care systems in addition to personal pain. [3]

This project aims to create a wearable air quality monitor that tracks an individual’s exposure to organic pollutants. To achieve this, a custom printed circuit board (PCB) will be designed and developed. This PCB will be integrated into a comfortable wristband for easy, on-the-go monitoring.

1.1 Sustainable Development Goals

In a historic move in 2015, the United Nations member states adopted the 2030 Agenda for a new era of sustainability. This comprehensive strategy delineates a route to attain financial prosperity while safeguarding the environment and its inhabitants.

With climate change a growing threat, the seventeen Sustainable Development Goals (SDGs) offer a clear path forward to ensure a sustainable future for generations to come.

This project contributes to some of the SDGs (Figure 1).

Figure 1. SDGs related to the project.

Goal 3: Good health and well-being, particularly targets 3.9 and 3.a. It aims to reduce mortality and diseases caused by air pollution by monitoring exposure and controlling tobaco.

Goal 9: Industry, innovation and infrastructure. Although the project itself may not directly create new research and development jobs, it aligns with SDG 9. By developing a novel low-power wearable device, it serves as a powerful example for target 9.5, which encourages the adoption of sustainable technological advancements. This

innovation has the potential to spark further development in wearable technology for environmental monitoring or even health applications. Additionally, the data collected by the device can be a valuable resource for scientific research on air pollution exposure. This data can help us understand the health impacts on different demographics and inform the creation of targeted strategies to mitigate pollution's effects.

Goal 11: Sustainable cities and communities. While this project won't directly create safe, inclusive green spaces for all (Target 11.7), the data it collects can indirectly contribute. The collected information can inform policy decisions that promote the development of accessible and equitable public spaces.

By aligning with these specific SDGs, the project takes a multifaceted approach to building a more sustainable future.

1.2 The *OnBREATHE* project

The project this work contributes to is called *OnBREATHE*: "Personal air quality monitoring and data digitalization to track chronic respiratory diseases." This ambitious proposal brings together a team of eighteen researchers with diverse expertise. They hail from four distinct research groups: the Pediatric, Nutrition and Development Research Unit (PEDINUR) at the Institute of Health Research Pere Virgili (IISPV), the Microsystems Nanotechnologies for Chemical Analysis (MINOS) and Chromatography and Environmental Applications (CROMA) groups from the University of Rovira i Virgili, and a collaborating group from the Biomedical Research Center for Respiratory Diseases Network (CIBERES) at "La Princesa" hospital in Madrid. This collaboration across various fields, from environmental health to pediatric pneumology, underscores the project's multifaceted approach to tackling chronic respiratory diseases.

OnBREATHE's core mission is to create and verify a groundbreaking wearable digital tool. This tool will track both air quality and health markers in adult and pediatric patients with chronic respiratory illnesses, allowing for personalized monitoring of their exposure to organic pollutants. This solution has two specific objectives:

1. **Develop a monitoring device** that will track a patient's exposure to air pollutants with two functionalities:
   1.1. **Offline characterization** of captured volatile organic compounds (VOCs) the patient encounters over a period.
   1.2. **Continuous monitoring** with a built-in microsensor that will track real-time air quality by measuring temperature, humidity, pressure, and VOCs concentration.
2. **Create a smartphone app and a software platform** that will record and combine exposure data with the patient's health information and disease progression.
   2.1. **Data collection** about their health status, activity levels and exposure and allowing to download the continuous microsensor data to patients.
   2.2. **Data analysis** and processing by the researchers of the collected parameters stored in the software platform.

This work focuses on objective 1.2, developing a wearable microsensor that continuously monitors air quality for patients. A custom PCB will be designed and built specifically to detect VOCs. The microcontroller will transmit data via Bluetooth Low Energy (BLE) – explained in detail in Chapter 2.3 – to a smartphone app developed by the onLean company. This app will then securely transmit the data over HTTP to a remote server, allowing researchers and clinical staff to access patients' information through a dedicated software platform (Figure 2).

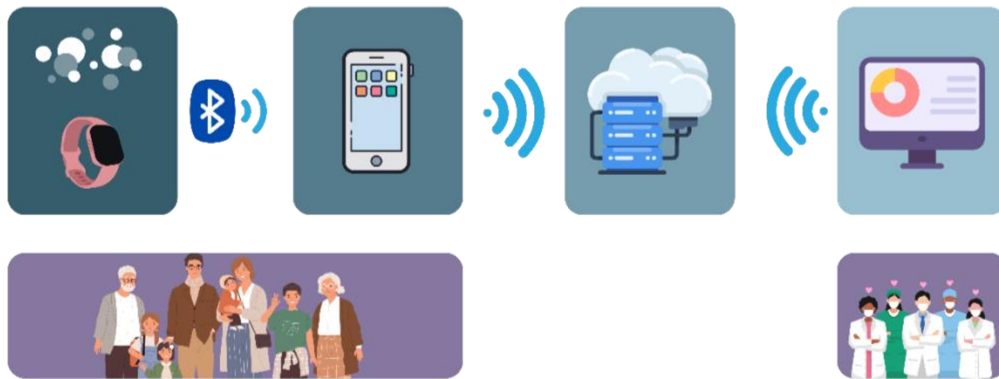


Figure 2. Graphical abstract of the OnBREATHE project.

1.3 Motivation

As mentioned on the introduction, public health experts are increasingly alarmed by the crisis of air pollution. Numbers underscore the urgent need for better ways to track and understand individual pollutants.

Traditional air quality monitoring stations provide a general picture, but they don't capture the variations in an individual's exposure throughout the day. My work aims to address this gap by creating a wearable device for personalized data collection. By continuously monitoring an individual's exposure, researchers hope to gain a deeper understanding of how our daily activities and environment affect our health.

This personal device goes beyond simple data collection, it also empowers individuals to take charge of their health. By tracking their exposure to pollutants, people can make informed decisions about their daily activities or environments. This information could be particularly valuable for those with health conditions worsened by air pollution.

The project also recognizes the economic overload of pollution-related illnesses. By enabling personalized monitoring, it could contribute to preventative measures and early intervention, potentially reducing the strain on healthcare systems.

1.4 Goals

This project aims to develop and validate a wearable air quality device for personalized monitoring that will wirelessly transmit real-time air quality data to a user's mobile phone app.

For a successful completion of the project, we will decompose the main goal into a series of specific subgoals:

1. Gain a comprehensive understanding of volatile organic compounds (VOCs).
2. Evaluate the feasibility of using Internet of Things (IoT) and Bluetooth Low Energy (BLE) technologies for efficient data transmission.
3. Understand the Inter-Integrated Circuit (I2C) communication protocol for seamless communication between the microcontroller and onboard components.
4. Study the selected components (STM32 microcontroller, NCP705MT33, BME680, LTC2942-1 and MCP73831) for optimal performance and power efficiency.
5. Develop robust and efficient software for the microcontroller to manage sensor data acquisition, processing, and communication with minimal power consumption.
6. Optimize the PCB design for miniaturization while maintaining functionality and reliability.
7. Validate and evaluate the functionality and performance of the wearable device under various conditions and analyze the results to identify areas for improvement.

Once the project is complete, all objectives will be reassessed to determine if they were met and identify key learnings from achieved results.

2 Theoretical framework

This chapter goes through the theoretical foundation for the development of the low-power wearable monitoring device. We will begin by exploring the target pollutants, volatile organic compounds, and semi-volatile organic compounds, in Chapter 2.1. Next, we delve into the communication technologies that will enable the device to transmit data, discussing the Internet of Things (IoT) concept in Chapter 2.2 and Bluetooth Low Energy (BLE) in Chapter 2.3. To facilitate communication between various sensors within the microcontroller, we will go through the Inter-Integrated Circuit (I2C) protocol in Chapter 2.4. Finally, Chapter 2.5 details the core components that will be integrated onto the Printed Circuit Board (PCB).

2.1 VOCs

While air harbors a multitude of contaminants, our primary concern lies in monitoring volatile organic compounds (VOCs).

Volatile organic compounds (VOCs) are gaseous pollutants that easily vaporize due to their high vapor pressure and low water solubility by some products or processes [4, 5]. VOCs readily transform into gases at low temperatures with their high volatility [6]. Among VOCs, some familiar examples include toluene, formaldehyde, and benzene.

Although VOCs are present in both indoor and outdoor spaces, investigations from the United States Environmental Protection Agency revealed that levels of approximately a dozen typical VOCs were two to five times greater inside homes than outside.

Common indoor sources of VOCs include paint, cleaning products, disinfectants, pesticides, tobacco smoke, and stove emissions. Outdoor sources encompass gasoline/diesel vehicles, wood burning, oil and gas extraction/processing, and industrial facilities. While both indoor and outdoor sources contribute to human health risks, outdoor sources additionally pose ecological threats (Figure 3). [7]

Inhalation of VOCs can irritate the respiratory system, damage the central nervous system and other organs, leading to nausea and breathing difficulties [4]. Moreover, sunlight-driven interactions between VOCs and nitrogen oxides (NO_x) generate harmful photochemical oxidants such as ozone, negatively impacting human health and the environment, with NO oxidation to NO_2 significantly increasing environmental damage [7].

To address the harmful effects of these pollutants, the Council of the European Union adopted Directive 1999/13/EC (superseded by Directive 2010/75/EC) in 1999. This legislation regulates the use of solvents in large quantities across various industries, including graphic design, pharmaceuticals, automobiles, and even small businesses like laundromats. Implemented in Spain through Royal Decree 117/2003, the directive has seen modifications at autonomous levels, though Catalonia hasn’t adopted any change.

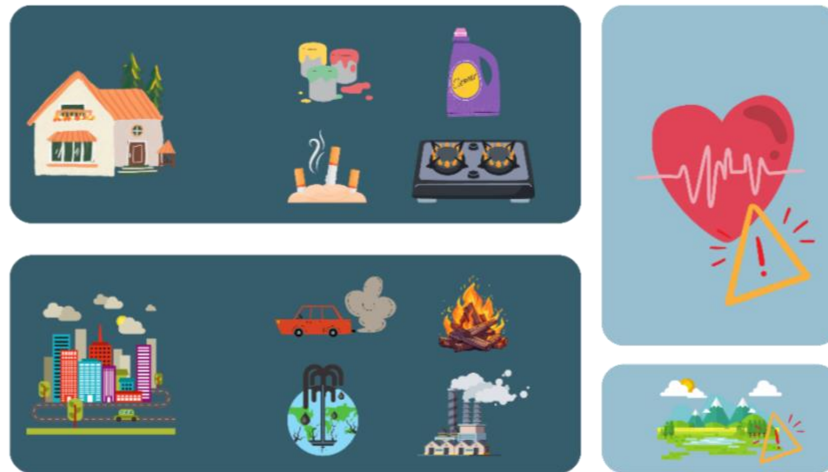


Figure 3. Main sources of VOCs.

2.2 Internet of Things

The term “Internet of Things” (IoT) describes the process of linking everyday objects – from lights in your home to fitness trackers on your wrist – to the internet, creating a network of smart devices that can even extend to entire cities. These objects can seamlessly exchange data without human input. To facilitate this communication, each device is assigned a unique identifier, forming the backbone of the interconnected IoT network.

This concept is mainly used to refer to devices that were never intended to be connected to the internet. Imagine a scenario where your smart thermostat retrieves your location data from your connected car. As you approach home, the thermostat automatically adjusts the temperature to your preference, ensuring a comfortable arrival even before you step through the door. This is just one example of the vast potential of IoT, where everyday objects are interconnected, creating a smarter and more automated world.

A wide range of industries, from automotive and healthcare to manufacturing, consumer electronics, and beyond, are actively exploring how to integrate IoT technology into their products, services, and operations (Figure 4).

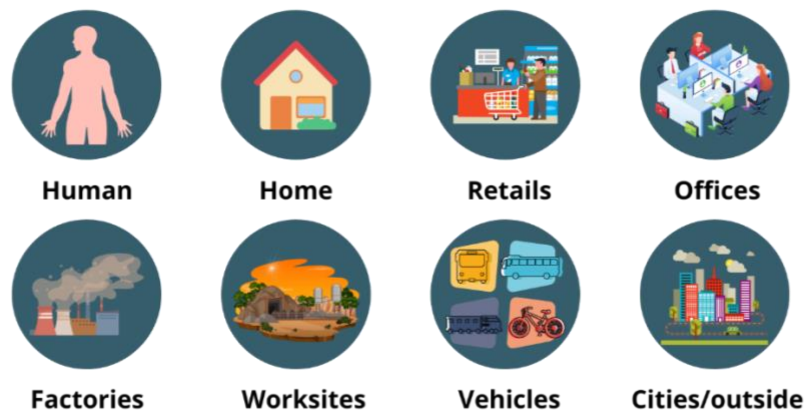


Figure 4. Environments of IoT applications (adopted from [8]**).**

When it comes to networks powering the IoT, there are two main categories that classify the various technologies used (Table 1). Short-range networks are idea for connection devices within a limited physical space, like a home or a building. On the other hand, long-range networks enable communication over wider distances, making them suitable for connecting devices spread across cities or even continents.

Type	Technology	Range
Low-power, short-range	Bluetooth	< 10 m
	NFC	< 4 cm
	Wi-Fi	< 30 m
	Z-Wave	< 100 m
	Zigbee	< 100 m
Low-power, wide-area	4G/5G	< 16 km
	5G	< 300 m
	LoRaWan	< 3 km

Table 1. Types of IoT networks.

Internet of Things serves as the backbone of this project, enabling seamless communication and data collection from the diverse components embedded within the wearable device. Furthermore, leveraging IoT enables secure and efficient transmission of the collected data to cloud platforms for in-depth analysis and visualization. Therefore, a strong understanding of IoT is crucial for ensuring the wearable's device successful implementation and effectiveness in monitoring the health impacts of pollution on individuals.

2.3 Bluetooth Low Energy

To enable low-power wireless data transmission to a mobile phone, the microcontroller will employ Bluetooth Low Energy (BLE) technology (as discussed in Chapter 1.2). While the selected microcontroller supports Zigbee, OpenThread, and BLE, BLE is the preferred choice due to its widespread adoption.

Bluetooth Low Energy (BLE) is a wireless communications technology developed by Nokia and Bluetooth SIG. Despite its perceived newness, BLE has been around since 2011, with the iPhone 4s being the first mobile phone to implement it. This low-power capability is crucial for the growth of the Internet of Things (IoT) movement, where many devices rely on battery power for extended operation [9].

The Bluetooth ecosystem emerged with classic Bluetooth, which aimed to bridge the gap between computing devices and communication. Initially focused on audio sharing, it quickly expanded to include applications like wireless printing and file transfer. This use cases required high bandwidth rates, leading to the development of Bluetooth Low Energy (BLE), which was optimized for ultra-low power operation (Table 2). BLE's design targets two key areas: enabling wireless functionality in devices traditionally lacking it and facilitating large-scale deployments due to its cost-effectiveness.

Feature	Bluetooth Classic	Bluetooth Low Energy
Power consumption	1 W	0.01 to 0.5 W
Data throughput	2.1 Mbit/s	0.3 Mbit/s
Range	100 m	< 100 m
Active slaves	Not limited	< 8
Current consumption	< 30 mA	< 15 mA

Table 2. Differences between Bluetooth Classic and Bluetooth Low Energy (adopted from [10]**).**

With a comprehensive understanding of BLE and its distinction from classic Bluetooth, we can now explore the protocols that structure and enable BLE communication.

The first protocol, the Generic Access Profile (GAP), governs the core functionalities of BLE: advertising for discoverability and establishing connections. This ensures your device is visible and communication proceeds smoothly.

BLE advertisement relies on well-defined roles established by the GAP protocol. These roles dictate how devices interact and discover each other. There are two primary role pairings:

- Broadcast – Observer: in this pairing, the broadcaster transmits advertisement packets containing data but doesn't allow connections. Contrariwise, the observer scans for these packets but doesn't initiate the connection itself.
- Central – Peripheral: these pairing forms the foundation of most BLE connections, including the one explored in this work. The central device actively searches for advertising messages broadcast by peripherals (Figure 5). If a suitable match is found, the central initiates a connection with the peripheral.

Figure 5. Illustrative example of BLE advertisement.

After the connection is established, devices exchange data following a set of rules called Generic Attribute Profile (GATT). GATT uses Services and Characteristics to categorize data (Figure 6). It relies on another set of rules, Attribute Protocol (ATT), to organize this data like a simple dictionary.

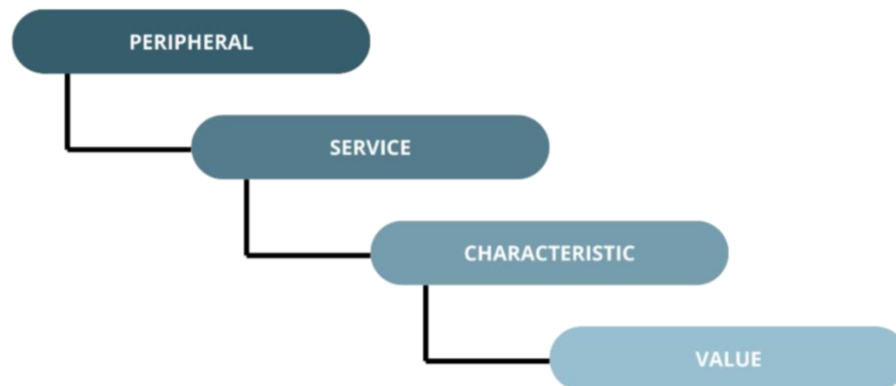


Figure 6. Generic Attribute Profile hierarchy.

Within the GATT framework, services act as containers for logically grouped data. Each service compasses specific data elements known as characteristics. To ensure unique identification, services are assigned Universally Unique Identifier (UUIDs). These UUIDs come in two variants: 16-bit for standardized BLE services and 128-bit for custom implementations.

The Bluetooth Developer Portal maintains a comprehensive registry of standardized BLE services. To leverage this standardized approach for the air quality monitoring device, we will employ the defined air quality service (0x0542).

GATT utilizes characteristics as the fundamental units of data exchange. Each characteristic holds a single data point and, similar to services, is identified by a unique UUID. Developers have the flexibility to leverage standardized characteristics defined by the Bluetooth SIG for interoperability across BLE devices and software. Notably, while standardized characteristics exist for basic environmental measurements like temperature or humidity, the Bluetooth SIG's registry currently lacks a characteristic specifically designed for Indoor Air Quality (IAQ) index.

A key distinction to remember regarding GATT and connections is their single-threaded nature. In simpler terms, a BLE peripheral can only maintain a single active connection with a central device (like at phone) at a time. Only through a connection can the central device transmit data to the peripheral and receive data back, fostering bidirectional communication.

For reliable and efficient data transmission, this work uses the Bluetooth Low Energy (BLE) technology. BLE offers significant advantages for wearable applications. It’s low power consumption ensures long battery life for the device, minimizing the need for frequent charging and maximizing user comfort. Additionally, BLE’s robust connection management allows for stable data transfer between the wearable and a smartphone, guaranteeing the integrity of the collected information.

2.4 Inter-Integrated Circuit

Developed by Philips in the early 1980s, the Inter-Integrated Circuit (I2C) protocol offers a low-cost and high-performance solution for peripheral device integration within embedded systems. Widely adopted as the go-to method, the I2C bus allows to connect and control a variety of components like temperature sensors, memory chips, and even small processors.

I2C allows multiples devices to connect with just two wires. This synchronous, two-way communication enables adding or removing devices without disruption others on the bus.

The I2C protocol relies on two key wires for communication: SDA carries the data, and SCL synchronized the data transfer with a clock signal (Figure 7). These wires stay high by default thanks to a connection to the power supply through pull-up resistors. Devices on the bus control communication by pulling the lines down when needed. Each device has a unique address and can act as either a sender (master) or receiver (slave), or even both. I2C stands out for being a multi-master bus. This means several devices can initiate communication, not just one central device.

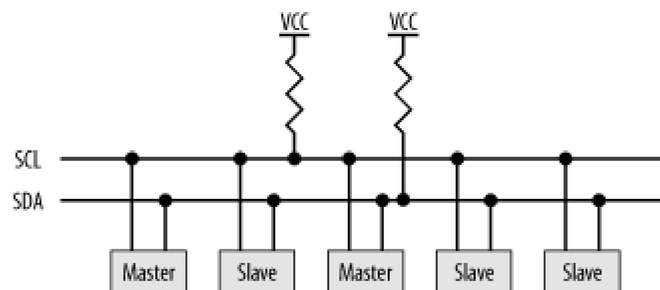


Figure 7. I2C block diagram [11].

The I2C bus ensures smooth communication by precisely coordinating the timing between SDA and SCL. In the idle state, both lines remain high. The I2C protocol starts with a signal called START. This is initiated by the master lowering the SDA line first, followed by the SCL line (Figure 8). This tells all the devices on the bus to pay attention because data is about to be transmitted. The very first bit of data is then sent while the clock line remains low.

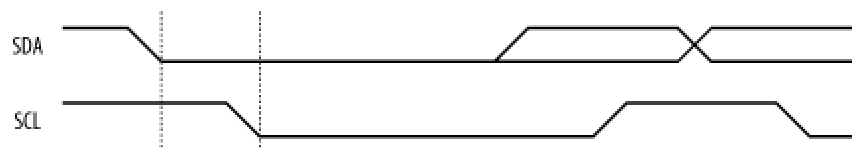


Figure 8. I2C START signal [11].

The data bit must stay unchanged until the SCL line goes low, indicating the end of the sampling period. Prior to SCL rising high once more, SDA moves on to the following bit.

At last, the bus ends with a STOP signal (Figure 9). SCL returns to high followed by SDA.

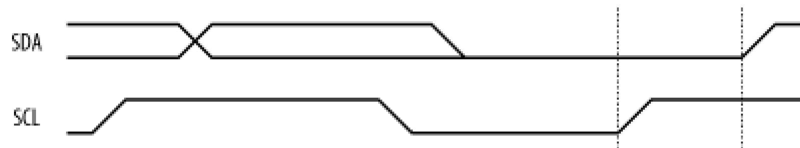


Figure 9. I2C STOP signal [11].

I2C ensures reliable data transfer by incorporating acknowledgements. Following the transmission of each byte, the master leaves control of the data line. Then, it generates an additional clock pulse. This causes the receiver to pull the data line low, acknowledging the byte (Figure 10).

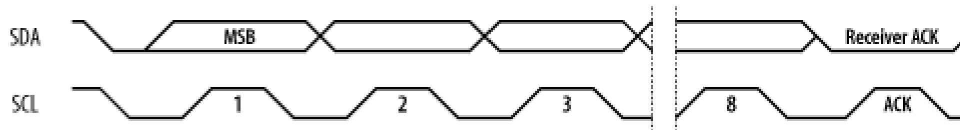
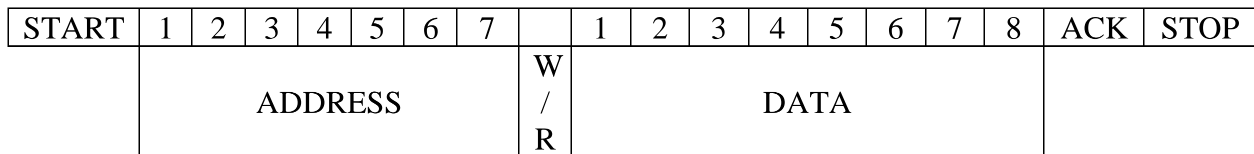


Figure 10. I2C byte acknowledgement signal [11].

As it was already mentioned, each device has a unique 7-bit identification address. Before data transmission begins, the master sends this address followed by a single bit to indicate the operation type: 1 for reading from the slave device or 0 for writing data to it (Figure 11).



START	1	2	3	4	5	6	7		1	2	3	4	5	6	7	8	ACK	STOP
	ADDRESS							W / R	DATA									

Figure 11. I2C packet.

2.5 Printed Circuit Board

A printed circuit board (PCB) can be defined as a platform for mounting and electrically connecting electronic components.

My involvement with the OnBREATHE project came after the PCB had already been designed and components selected. Thence, in this subchapter, we will delve into a detailed analysis of the selected components and their key characteristics.

While a detailed exploration of the design software is reserved for Chapter 4.1, Autodesk's Eagle was used for PCB design and production files were sent to Eurocircuits for manufacturing.

Despite comprising numerous components, the PCB can be logically categorized into three primary groups (Figure 12):

- A **microcontroller unit** (MCU). The wearable's brain, responsible for orchestrating all device operations.
- An **environmental monitoring sensor**. From Bosch Sensortec manufacturer, the BME680 offers continuous digital outputs for various air quality parameters.
- **Battery-related components**. They rule the charging process initiated at the pogo pins, ensuring the delivered voltage aligns with the PCB's operational range.

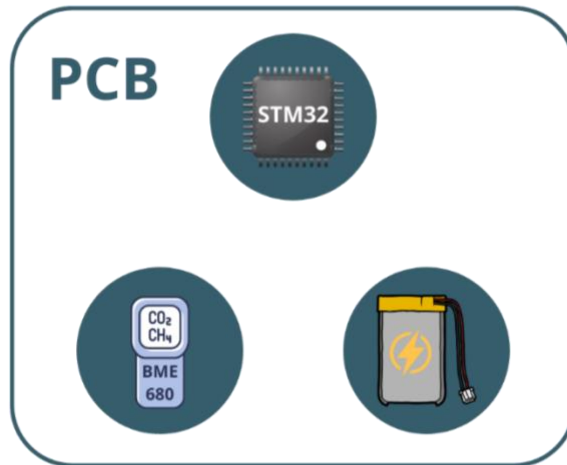


Figure 12. High-level overview of the PCB.

After establishing a comprehensive overview of the PCB's architecture, we can highlight the key characteristics of each component (Table 3), paving the way for an in-depth examination of each element's function.

Manufacturer	Component	Variable	Unit	Accuracy	Cost
STMicroelectronics	STM32WB5MMG H6TR	-	-	-	11 €
Bosch Sensortec	BME680	Breath-VOC equivalents	ppm	-	10 €
		CO_2 equivalents	ppm	-	
		Humidity	% r. H.	± 3	
		Pressure	hPa	± 0.6	
		Temperature	° C	± 1.0	
Analog Devices	LTC2942-1	Electric Charge	C	1 %	6 €
		Voltage	mV	± 9	
		Temperature	K	± 5	
Microchip Technology	MCP73831	-	-	-	1 €
ON Semicondutor	NCP705MT33	Voltage	V	± 2 %	1 €

Table 3. Characteristics of the components used in the PCB.

2.5.1 Microcontroller (STM32WB5MMG)

This STMicroelectronics component is an ultra-low-power microcontroller, facilitating energy harvesting and extending battery life significantly. Its compacts dimensions of 7.3 x 11 x 1.342 (width x height x depth) mm make it particularly noteworthy for the miniaturization discussions in Chapter 4.

The module offers comprehensive wireless connectivity, supporting Bluetooth Low Energy, Zigbee, OpenThread, dynamic/static concurrent modes and 802.15.4 protocols. It also incorporates the following third-party components (Figure 13):

- High/Low Speed External Oscillators (HSE / LSE)
- Switched-Mode Power Supply (SMPS)
- Antenna
- Integrated Passive Devices (IPD)

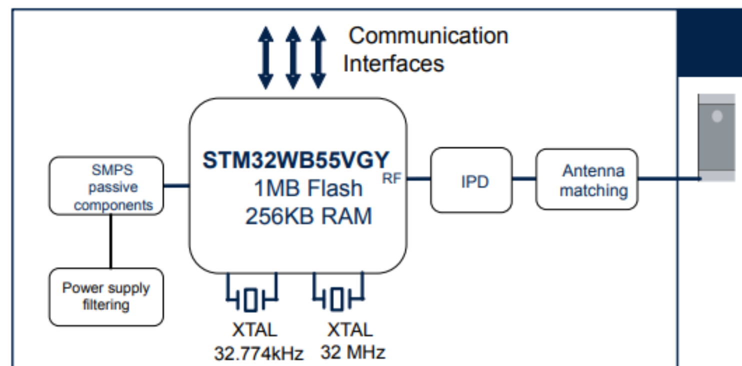


Figure 13. Block diagram of the STM32WB5MMG module [12]**.**

By leveraging the module's external pins, the system gains access to essential peripherals:

- USART
- LPUART (low power)
- SPI
- I2C
- Ultra-low power timers

According to the datasheet, the microcontroller requires a voltage supply between 1.71 V and 3.6 V. As the battery charger provides 5 V, a low-dropout regulator (LDO), detailed in Chapter 2.5.2, will step down the voltage to a suitable 3.3 V level. To optimize power efficiency, the microcontroller offers six low-power modes (

Table 4).

Mode	CPU	Flash	Clocks	Peripherals	Wake-up source	Consumption
Run (default, not low power)	✓	ON	Any	All	N/A	107 µA/MHz
LPRun	✓	ON	Any	All	N/A	103 µA/MHz
Sleep	✗	ON	Any	All	Any interrupt	46 µA/MHz
LPSleep	✗	ON	Any	All except RF	Any interrupt	45 µA/MHz
Stop 0	✗	OFF	LSE, LSI,	RF, RTC, LPUART,	Not-frozen peripherals	100 µA

			HSE, HSI	I2Cx, LPTIMx. Other peripherals frozen		
Stop 1	✗	OFF	LSE, LSI, HSE, HSI	RF, RTC, LPUART, I2Cx, LPTIMx. Other peripherals frozen	Not-frozen peripherals	9.6 µA
Stop 2	✗	OFF	LSE, LSI	RF, RTC, LPUART, I2C3, LPTIM1. Other peripherals frozen	Not-frozen peripherals	2.1 µA
Standby	✗	OFF	LSE, LSI	RF, RTC. Other peripherals off	RF, RTC	0.60 µA
Shutdown	✗	OFF	LSE	RTC. Other peripherals off	RTC	0.315 µA

Table 4. Low-power mode comparison for the STM32WB5MMG (adopted from [13]**).**

The selected microcontroller complies with various European regulatory standards, including CE, RED, RoHS, and REACH. Additionally, its Bluetooth Low Energy module holds a RF-PHY certification by the BLE SIG organization. These certifications facilitate future market approval for the wearable.

2.5.2 LDO regulator (NCP705MT33)

The PCB incorporates a low-dropout (LDO) regulator. This component efficiently converts a higher input voltage to a precisely regulated lower output voltage. In our application, the LDO regulator receives a 5 V from the charger connector and steps it down to a stable 3.3 V supply for the microcontroller and the environmental sensor (Figure 14).

Due to its ultra-low noise output and high Power Supply Rejection Ratio (PSRR), the selected LDO regulator from ON Semiconductor is ideally suited for wireless communication applications.

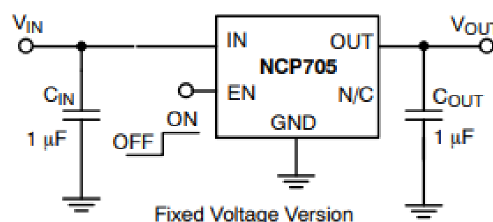


Figure 14. NCP705 Typical Application circuit [14]**.**

2.5.3 Gas sensor (BME680)

At the core of the wearable alongside the microcontroller lies the BME680 sensor. This digital sensor integrates four key environmental measurement: gas, humidity, pressure, and temperature. Its compact size (3 mm x 3 mm and 1 mm height) in a metal-lid package, coupled with low power consumption, make the BME680 an ideal choice for our application due to its miniaturized design and minimal impact on battery life.

The BME680 offers communication flexibility through both SPI and I2C interfaces. As explained in Chapter 2.4, we opted for the I2C bus.

Powering the sensor aligns with the LDO regulator's output voltage discussed in Chapter 2.5.2. With an operating range of 1.71 V to 3.6 V, the BME680 is compatible with the LDO's 3.3 V output. Notably, the sensor presents low current consumption, drawing only 3.7 μA at 1 Hz operation or a mere 0.15 μA in sleep mode.

The BME680 prioritizes low-power operation by default, starting in sleep mode. To initiate a measurement cycle encompassing all four sensors (temperature, pressure, humidity, and gas), the user can trigger a transition to forced mode. Following the data acquisition, the sensor automatically reverts to sleep mode, minimizing power consumption until the next user-initiated measurement. This sleep-on-wake cycle ensures efficient operation, balancing the power conservation with data collection needs.

To unlock the full potential of the BME680 sensor, we will leverage the Bosch Software Environmental Cluster (BSEC) solution alongside the BME680 sensor API (Table 5), as detailed in Chapter 3.1.2. BSEC incorporates sophisticated algorithms that extend the sensor's capabilities beyond raw data collection. Notably, BSEC enables applications like indoor air quality monitoring by analyzing the BME680's gas readings. Furthermore, these algorithms provide crucial data correction functionalities, including humidity compensation, baseline correction, and long-term drift mitigation for the gas sensor signal.

BME680	BSEC
Temperature	Sensor-compensated temperature
Humidity	Sensor-compensated relative humidity
Gas resistance	Sensor-compensated gas sensor resistance
Humidity	IAQ
	CO2 equivalents
	b-VOC equivalents

Table 5. Comparison of BME680 and BSEC output parameters.

The BME680 utilizes a metal-oxide (MOX) sensing element. MOX sensors rely on changes in electrical resistance due to the adsorption of target gases on a heated metal oxide layer. This adsorption process can involve oxidation or reduction reactions on the surface. Therefore, the BME680 exhibits a broad sensitivity to volatile organic compounds (VOCs) and many other common indoor air pollutants, with a notable exception being carbon dioxide (CO_2). Unlike selective gas sensors that target specific molecules, the BME680 offers the advantage of measuring the total VOCs concentration

in the surrounding air. This feature allows the BME680 to detect a wide range of indoor air quality issues, including outgassing from building materials (paint, furniture), elevated VOC levels due to cooking activities, food consumption, and human exhalation (breath and sweat).

As previously discussed, the BME680 outputs raw gas sensor resistance values that change with varying VOC concentrations. However, this raw signal is susceptible to influences beyond just VOCs, including humidity levels. To address this limitation and provide a more meaningful representation of air quality, the BSEC employs smart algorithms. These algorithms transform the raw resistance values into an air quality (IAQ) index. This metric offers a more accurate assessment of indoor air quality by effectively compensating for environmental factors that can skew the raw sensor data.

The BSEC-derived air quality (IAQ) index ranges from 0, indicating clean air, to 500, representing heavily polluted air (Figure 15). This numerical scale provides a user-friendly representation of indoor air quality. The algorithms consider the sensor's typical operating environment to ensure consistent IAQ readings. The calibration process leverages recent measurement history to establish baselines. This approach ensures that an IAQ value of approximately 50 corresponds to "typical good", while an IAQ of approximately 200 reflects "typical polluted" air.

IAQ Index	Air Quality	Impact (long-term exposure)	Suggested action
0 – 50	Excellent	Pure air; best for well-being	No measures needed
51 – 100	Good	No irritation or impact on well-being	No measures needed
101 – 150	Lightly polluted	Reduction of well-being possible	Ventilation suggested
151 – 200	Moderately polluted	More significant irritation possible	Increase ventilation with clean air
201 – 250	Heavily polluted	Exposition might lead to effects like headache depending on type of VOCs	optimize ventilation
251 – 350	Severely polluted	More severe health issue possible if harmful VOC present	Contamination should be identified if level is reached even w/o presence of people; maximize ventilation & reduce attendance
> 351	Extremely polluted	Headaches, additional neurotoxic effects possible	Contamination needs to be identified; avoid presence in room and maximize ventilation

Figure 15. BSEC-Derived Air Quality Index classification, impact, and suggested actions [15].

2.5.4 Battery gauge (LTC2942-1)

The LTC2942-1 is a battery management integrated circuit (IC) that functions as a fuel gauge. It accomplishes this by measuring the accumulated charge and discharge currents, enabling precise estimation of the remaining battery capacity.

To achieve this precise estimation, the LTC2942-1 employs a multi-faceted approach. It integrates a coulomb counter, which tracks current flow through an internal resistor connected between the battery and its load. This provides a measure of accumulated charge. Additionally, the IC measures both battery voltage and its own

temperature using a high-precision converter. Charge, voltage and temperature are stored within the chip's memory and can be accessed via I^{2}C communication interface. For enhanced safety and monitoring, the LTC2942-1 allow to program high and low thresholds for each measured parameter. If a threshold is breached, the device triggers an alert through the I^{2}C protocol or by setting a flag within its internal status register.

The LTC2942-1 comes in a compact 2 mm x 3 mm footprint and uses a six-pin package (Figure 16). The SENSE+ pin established the connection point for the charger, while the SENSE- pin connects directly to the single-cell Li-Ion battery. Communication with the chip is achieved through the I^{2}C bus using the SDA and SCL pins. Lastly, the AL/CC pin is used for alert input and GND for system grounding.

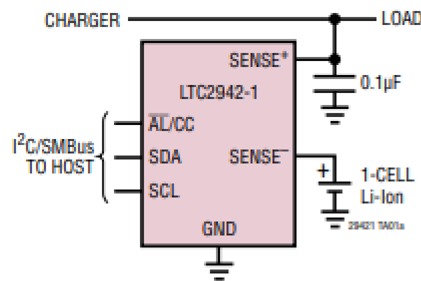


Figure 16. LTC2942-1 typical application circuit [16].

Configuration of the LTC2942-1's functionalities is achieved through a dedicated control register (Figure 17). In our implementation, the ADC mode is set to automatic, triggering battery status checks every second. The AL pin is configured as an input as it connects to the battery charger sensor, detailed in Chapter 2.5.5.

ADDRESS	NAME	REGISTER DESCRIPTION	R/W	DEFAULT
00h	A	Status	R	See Below
01h	B	Control	R/W	3Ch
02h	C	Accumulated Charge MSB	R/W	7Fh
03h	D	Accumulated Charge LSB	R/W	FFh
04h	E	Charge Threshold High MSB	R/W	FFh
05h	F	Charge Threshold High LSB	R/W	FFh
06h	G	Charge Threshold Low MSB	R/W	00h
07h	H	Charge Threshold Low LSB	R/W	00h
08h	I	Voltage MSB	R	XXh
09h	J	Voltage LSB	R	XXh
0Ah	K	Voltage Threshold High	R/W	FFh
0Bh	L	Voltage Threshold Low	R/W	00h
0Ch	M	Temperature MSB	R	XXh
0Dh	N	Temperature LSB	R	XXh
0Eh	O	Temperature Threshold High	R/W	FFh
0Fh	P	Temperature Threshold Low	R/W	00h

R = Read, W = Write, XX = unknown

BIT	NAME	OPERATION	DEFAULT
A[7]	Chip Identification	0: LTC2942-1 1: LTC2941-1	0
A[6]	Reserved	Not Used.	0
A[5]	Accumulated Charge Overflow/Underflow	Indicates that the value of the accumulated charge hit either top or bottom.	0
A[4]	Temperature Alert	Indicates one of the temperature limits was exceeded.	0
A[3]	Charge Alert High	Indicates that the accumulated charge value exceeded the charge threshold high limit.	0
A[2]	Charge Alert Low	Indicates that the accumulated charge value dropped below the charge threshold low limit.	0
A[1]	Voltage Alert	Indicates one of the battery voltage limits was exceeded.	0
A[0]	Undervoltage Lockout Alert	Indicates recovery from undervoltage. If set to 1, a UVLO has occurred and the content of the registers is uncertain.	X

Figure 17. LTC2942-1's register map and control register [16].

2.5.5 Battery charger (MCP73831)

The MCP73831 is a compact, integrated circuit designed for managing the charging of lithium-ion batteries (Figure 18). This eight-pin device simplifies the charging process by incorporating essential control functions. The V_{DD} pins (1 and 2) provide the 5 V power supply from the charger connector, while the V_{BAT} pins (3 and 4) connect directly to the battery's positive terminal. The STAT pin (typically pin 5) serves as an input, receiving a signal from the LTC2942's alert output to terminate the charging cycle. The remaining pins, V_{SS}, EP (exposed pad typically tied to ground), and PROG (programming), are grounded for proper operation. Notably, a 10 kΩ resistor (R_{PROG}) connected in series with the PROG pin enables programming of the charging current, ensuring safe and efficient battery charging.

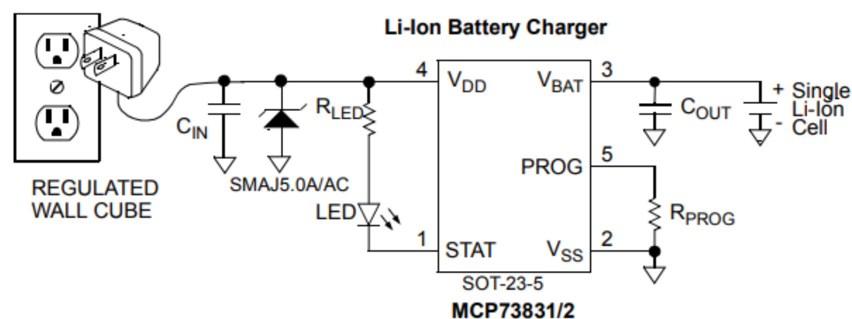


Figure 18. MCP73831 typical application circuit [17].

The charging process involves five stages to ensure optimal battery health and safety (Figure 19):

- **Shutdown mode**. The MCP73831 enters this mode when the voltage difference between the input supply voltage (V_{DD}) and the battery voltage (V_{BAT}) falls below a threshold of 50 mV. The device resumes operation only when the V_{DD} voltage rises at least 150 mV above V_{BAT}.
- **Preconditioning mode**. The preconditioning mode functionality is not available in our chosen MCP73831 variant.
- **Fast charge mode**. The component delivers a controlled charging current, typically set to 100 mA in our design, to the battery. This current is programmed using a single resistor connected between the PROG and VSS (ground) pins with a value of 10 kΩ in our case. This mode persists until the battery voltage (V_{BAT}) reaches the regulation voltage (V_{REG}), which is pre-set at 4,20 V for our chosen configuration.
- **Constant voltage mode**. Upon reaching the regulation voltage (V_{REG}) at the V_{BAT} pin, the MCP73831 automatically transitions to constant voltage regulation mode. This pre-set V_{REG} value is maintained with a high degree of accuracy (± 0.75% tolerance) to prevent battery overcharge. In this mode, the MCP73831 regulates the charging current, gradually reducing it as the battery approaches full capacity.
- **Charge complete mode**. The charging cycle reaches completion within constant voltage mode when the average charging current falls below a predetermined threshold. To prevent transient load fluctuations from triggering premature termination, the MCP73831 incorporates a 1 ms filter on the

termination comparator. Once the average current falls below this time threshold, the charging current is permanently terminated, and the device enters this mode.

Even after entering the charge complete mode, the MCP73831 stays alert by continuously monitoring the battery voltage (V_{BAT}). If the voltage falls below a pre-set recharge threshold, the device automatically initiates a new charging cycle. This ensures the battery remains sufficiently charged to power the load.

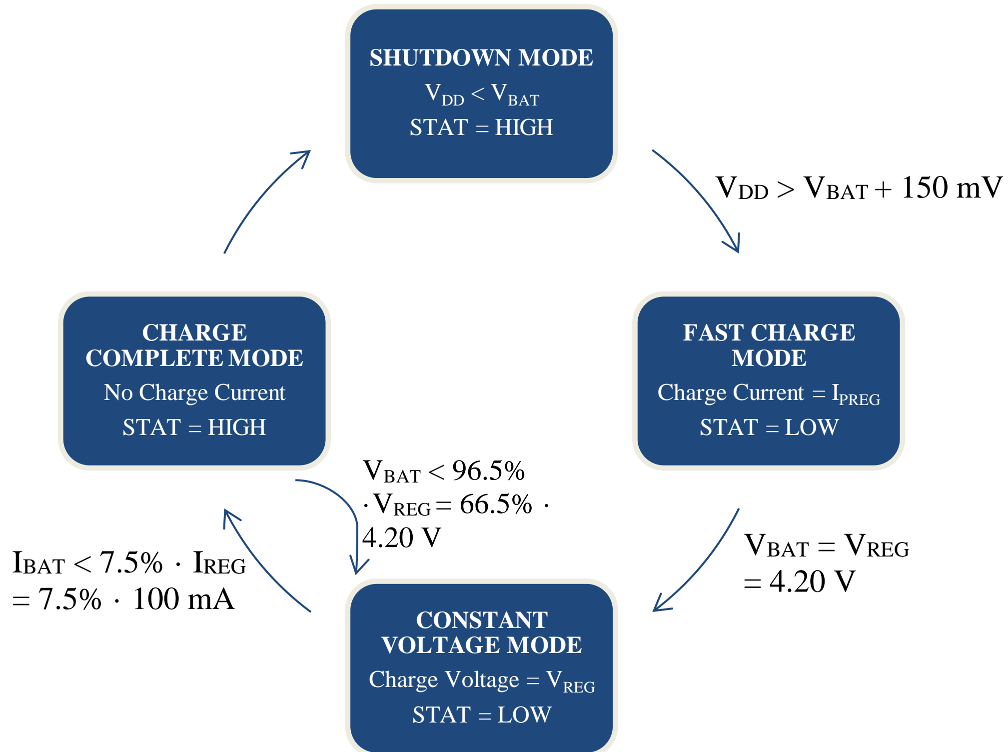


Figure 19. MCP73831 operating state diagram (adopted from [17]**).**

The STAT pin serves as a communication interface between the MCP73831 and the LTC2942-1 gauge. This pin operates in three distinct states to convey charging status:

- **High impedance**. This state signifies either shutdown mode or the absence of a battery. No current flows through the STAT pin in this condition.
- **Logic low**. During active charging, encompassing both fast charge and constant voltage modes, the STAT pin is driven low. This informs the LTC2942-1 gauge about the ongoing charging process.
- **Logic high**. A logic high on the STAT pin indicates the completion of the charging cycle, allowing the LTC2942-1 to update the battery status accordingly.

In short, the MCP73831 offers a compact and efficient solution for lithium-ion battery charging by integrating essential control functions. It effectively manages the charging process through precisely controlled current and voltage regulation. The clear communication between the MCP73831 and the LTC2942-1 gauge via the STAT pin ensures optimal battery health and safety throughout the charging cycle. This comprehensive approach guarantees safe, efficient, and reliable battery operation within the wearable.

3 Software Development

This chapter delves into the software development process for the Printed Circuit Boards (PCB), focusing on their integration with various hardware components. The chapter details essential aspects like utilizing a specific programming IDE to create and the develop the code (Chapter 3.1). It then progresses to discuss the configuration of the Inter-Integrated Circuit (I^2C) protocol for communication with external sensors (Chapter 3.1.1), exemplified by the BME680 sensor used for environmental data acquisition. Subsequently, the implementation of software routines for reading this environmental data from the BME680 sensor is explored (Chapter 3.1.2). Wireless data transmission via Bluetooth Low Energy (BLE) using the STM32 microcontroller is then examined (Chapter 3.1.3). Finally, the chapter discusses the integration of the LTC2942-1 gauge for effective power management through low-power mode operation, ensuring optimal battery life (Chapter 3.1.4).

3.1 STM32CubeIDE

This project leverages STM32CubeIDE, a dedicated integrated development environment (IDE). STM32CubeIDE provides a robust C/C++ development platform that allows efficient writing and compiling of code specifically tailored for STM32 microcontrollers. This support ensures compatibility and simplifies the development process.

Many microcontrollers, including the STM32 used in this project, come equipped with various built-in hardware components known as peripherals. STM32CubeIDE provides a user-friendly interface for configuring these peripherals, allowing developers to easily set them up for specific functionalities within their project. After selecting the desired peripherals, the IDE can automatically generate initialization code, saving developers significant time and effort. This reduces the risk of errors in low-level code and allows developers to focus on the core functionality of their project.

Beyond its development features, the IDE offers a valuable suite of debugging tools. This suite empowers developers to pinpoint and rectify errors within the code base. Features like register view, memory inspection, and variable watch enable fine-grained analysis of the program's state during execution.

STM32CubeIDE requires a free STMicroelectronics account for download and access to it. After installation and login, developers are greeted by a welcome screen (Figure 20). The manufacturer offers a comprehensive suite of developer resources, including video tutorials, user manuals, and wiki pages. However, this work will focus solely on the features essentials to achieving the project's specific goals.

We begin by creating a new STM32 project. To achieve this, navigate to the File menu, select New, and then choose STM32 Project (Figure 21). This action prompts STM32CubeIDE to communicate with its servers and retrieve the latest information on available STM32 models and example projects. Following this data retrieval, a project creation wizard launches.

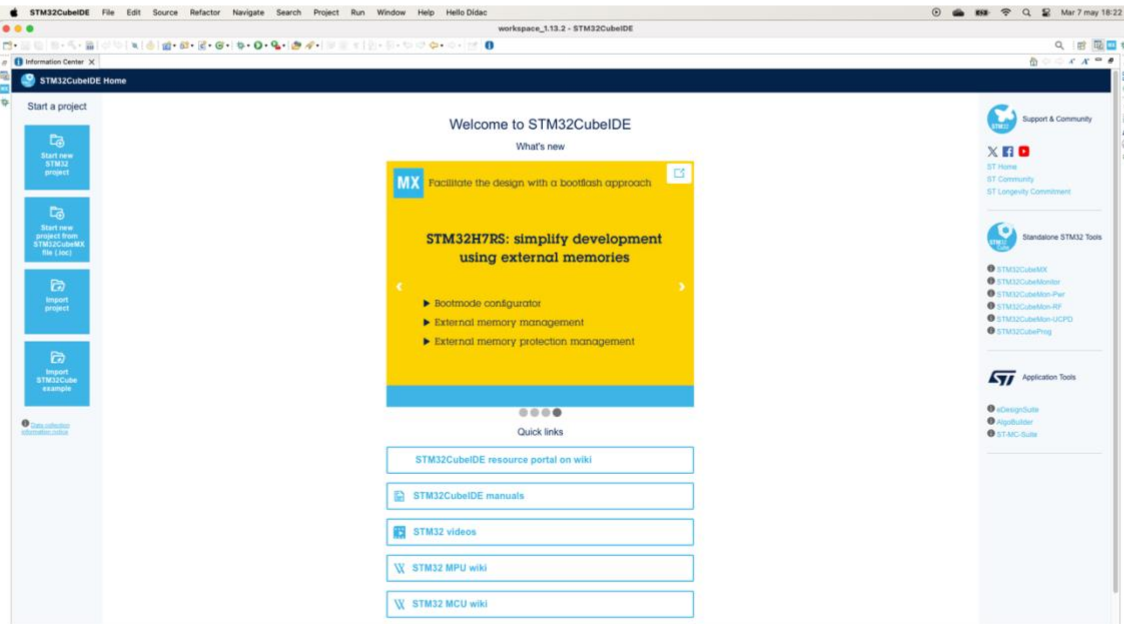


Figure 20. STM32CubeIDE welcome page.

Within the wizard, it asks to select the specific microcontroller you intend to use. In this instance, the microcontroller model is the STM32WB5MMGH6TR (Figure 21). After confirming the selection, proceed by clicking the Next button.

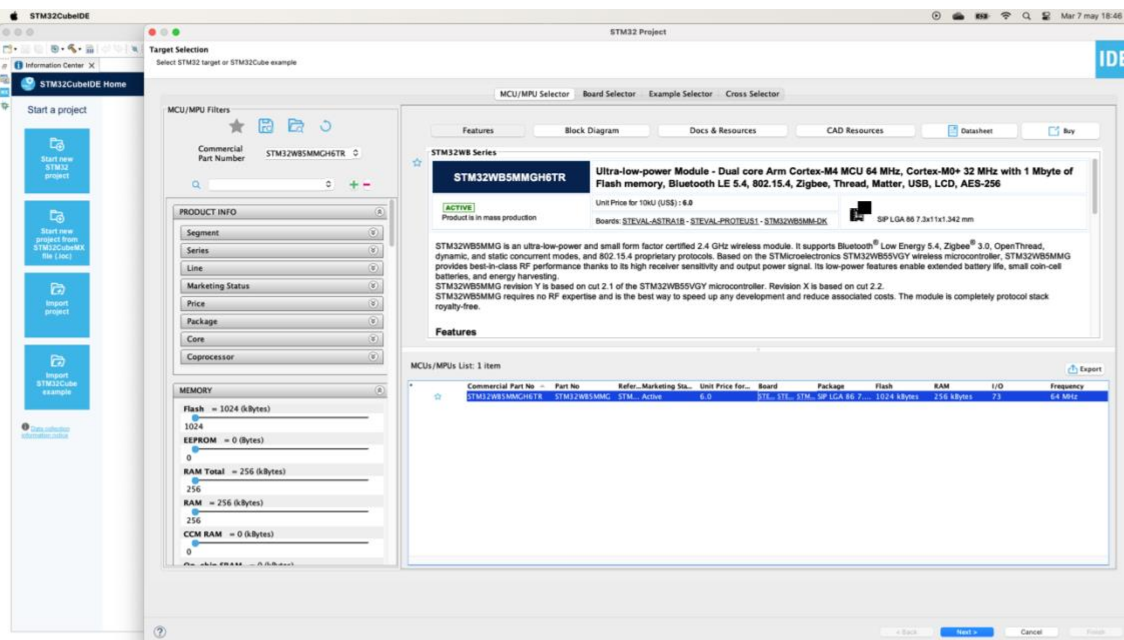


Figure 21. STM32CubeIDE target selection screen.

The subsequent step involves assigning a suitable name for the project (Figure 22). Additionally, the wizard prompts you to choose the development language (C in this case), the target binary type (executable), and the project type (STM32CubeIDE).

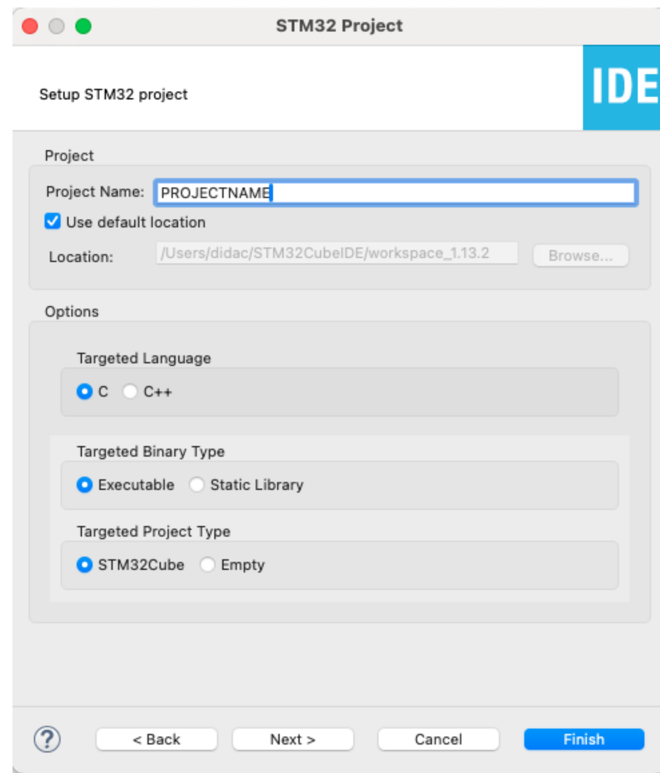


Figure 22. STM32CubeIDE project creation last step.

Following project creation, STM32CubeIDE launches the Development Configuration Tool (Figure 23). While presented as a visual editor, the configuration data is actually stored in a file with the .ioc extension. The interface is comprised of four primary sections: Pinout & Configuration, Clock Configuration, Project Manager and Tools.

For the purposes of this project, we will focus on the Pinout & Configuration section. This section allows to enable and configure the peripherals we will be using, specifically the I2C and BLE interfaces. A detailed exploration of these configurations will be provided in Chapters 3.1.1 and 3.1.3, respectively.

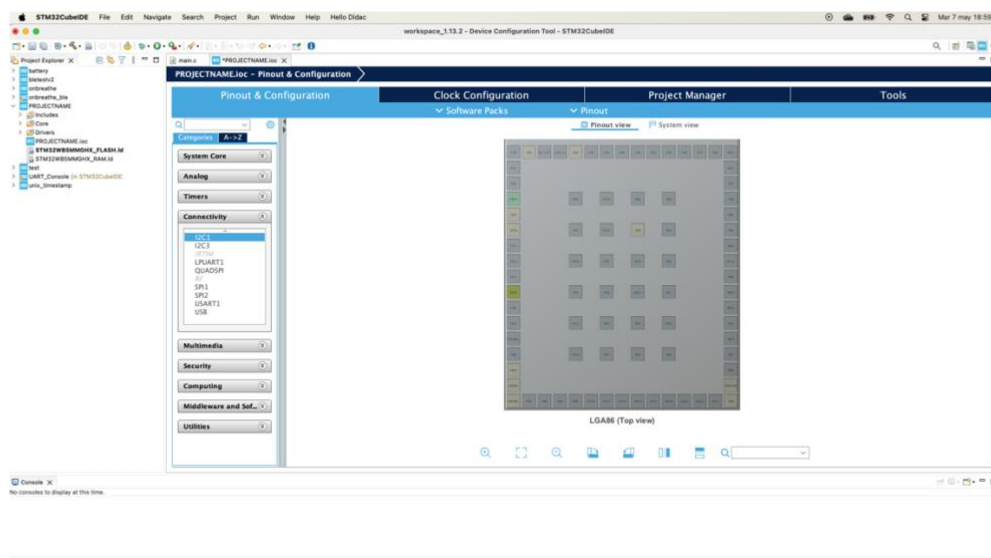


Figure 23. STM32CubeIDE Development Configuration Tool screen.

Having established a new project, the next step is to configure the connection between the development environment and the physical target hardware. This connection facilitates programming the microcontroller on the PCB (Printed Circuit Board) and enables debugging functionalities.

For this purpose, STM32CubeIDE integrates with STLINK devices, the official in-circuit programmer tool offered by STMicroelectronics, the manufacturer of the STM32 microcontrollers. The specific model used in this project is the STLINK-V3SET.

The STLINK-V3SET programmer offers versatile communication protocols for interacting with STM32 microcontrollers (Figure 24). These protocols include SWIM (Single Wire Interface Module), JTAG (Joint Test Action Group), and SWD (Serial Wire Debug). For this project, we will leverage the SWIM protocol due to its efficient use of only six pins.

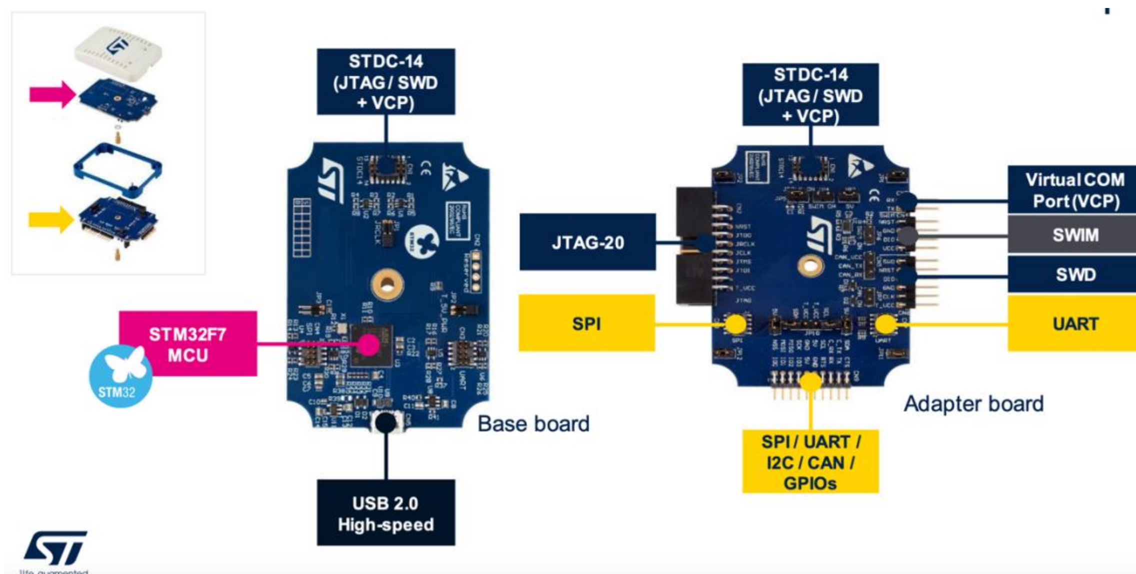


Figure 24. Close-up of the STLINK-V3SET highlighting the various communication interfaces it supports [18].

The SWIM interface requires connections to the following PCB pins via a pogo pin clip clamp:

- VCC (3.3V): provides power to the microcontroller.
- SWCLK (clock signal): synchronizes data transfer between the STLINK-V3SET and the microcontroller.
- SWDIO (data input/output): bi-directional data line for communication.
- NRST (reset): initiates a microcontroller reset.
- GND (Ground): provides a common ground reference for the connection.

Following successful wiring of the PCB and STLINK-V3SET programmer, the programmer should be connected to a computer using a USB Type-A to Micro-B cable (Figure 25). Upon successful connection and power delivery, the STLINK-V3SET's PWR LED and COM LED should illuminate.

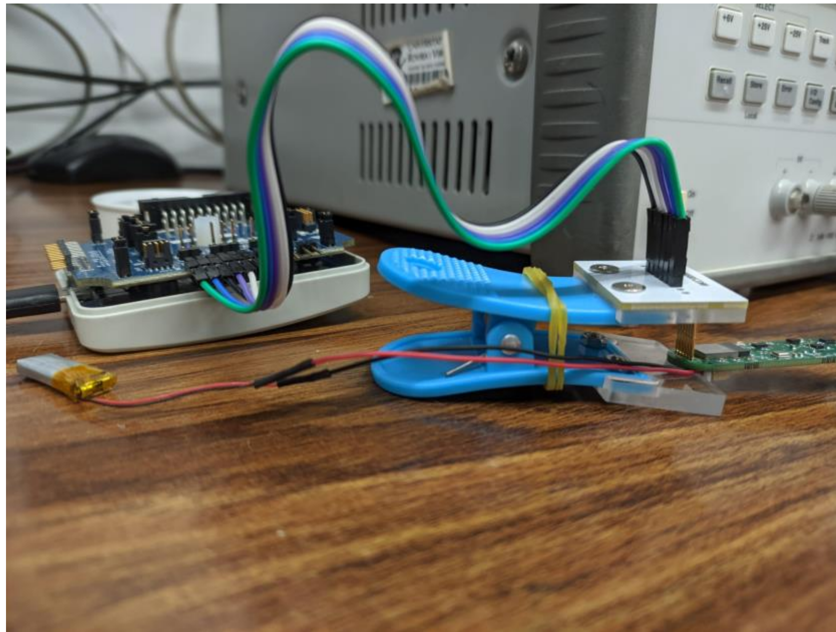


Figure 25. Connection of the STLINK-V3SET programmer to the PCB using the clip clamp.

To ensure compatibility and access to the latest features, we must consider verifying and updating the STLINK-V3SET firmware through STM32CubeIDE. To do that, we navigate to the Help menu and select ST-Link Upgrade (Figure 26). If connected properly, the STLINK-V3SET should appear in the list. Click on the device and select "Open in Update Mode" to initiate the upgrade process.

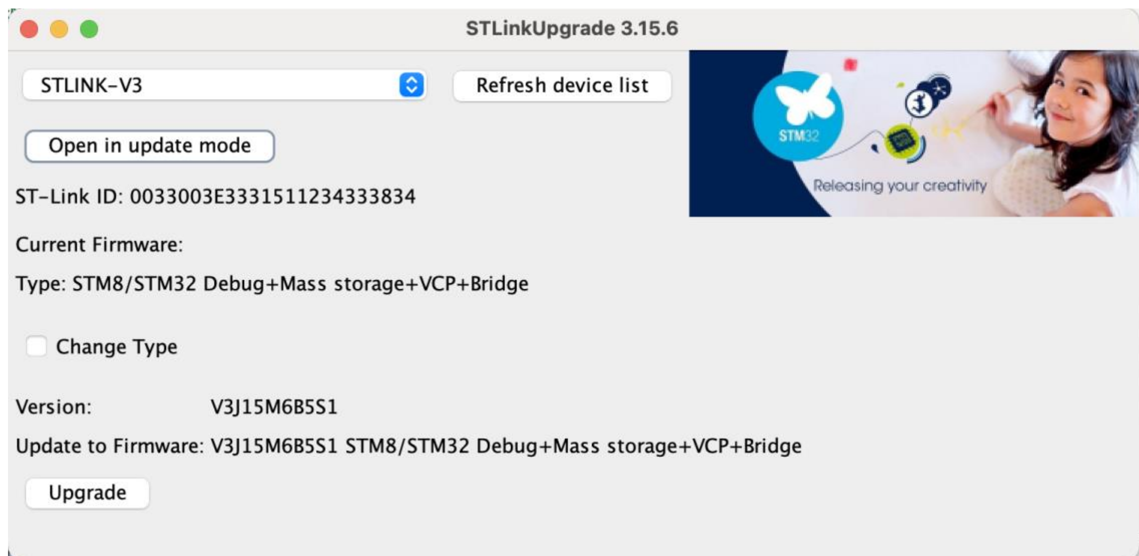


Figure 26. STLinkUpgrade screen.

Having successfully completed the hardware setup procedures, including project creation, STLINK-V3SET connection, and potential firmware update, we are now prepared to proceed with software development within the STM32CubeIDE environment.

3.1.1 Inter-Integrated Circuit

Building upon the details presented in Chapter 2.4, the Inter-Integrated Circuit (I2C) interface plays a critical role in this project. It facilitates communication between the microcontroller and essential peripherals, including the BME680 gas sensor and the LTC2942-1 battery gauge.

While our STM32 microcontroller offers two I2C interfaces, the schematics depict all peripherals connected to PB8 (SCL) and PB9 (SDA). Consulting the datasheet (DS11929), we find that PB8 and PB9 correspond to I2C1_SCL and I2C1_SDA, respectively. PB10 and PB11, which map to I2C3_SCL and I2C3_SDA, remain unconnected.

The I2C1 interface is enabled within the microcontroller's configuration tool, located under the “Connectivity” category. The mode is then selected as “I2C”. Verification of pin assignments can be performed within the tool, confirming that PB8 and PB9 are designated as I2C1_SCL and I2C1_SDA, respectively (Figure 27).

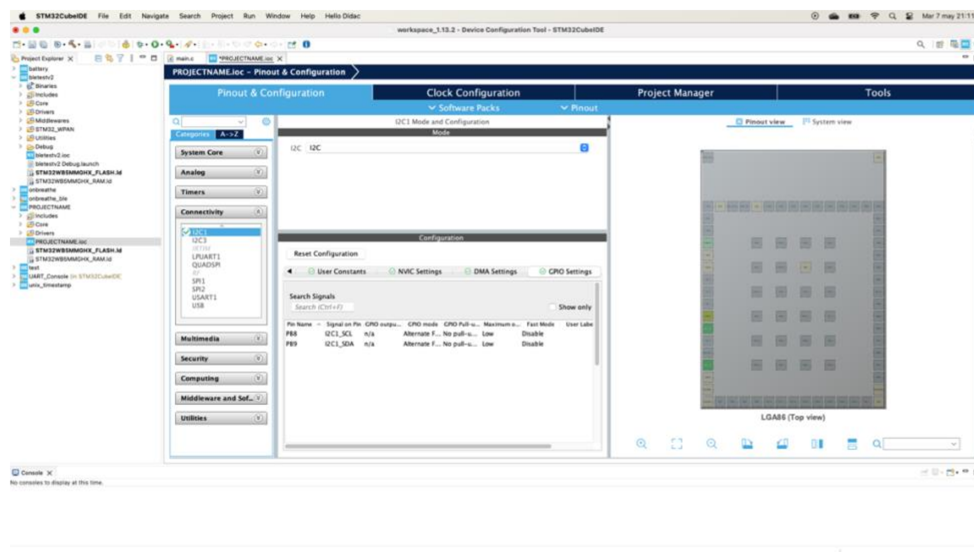


Figure 27. STM32CubeIDE I2C1 GPIO settings.

Following configuration, the project within the development tool must be saved (Figure 28). This action prompt the tool to generate code specifically tailored to utilize the I2C1 interface (Figure 29).

Figure 28. STM32CubeIDE toolbar.

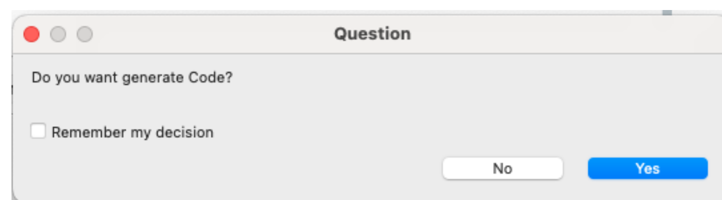


Figure 29. STM32CubeIDE confirmation prompt for generating code.

3.1.2 Gas sensor (BME680)

Complementing the microcontroller, the BME680 sensor plays a vital role in the wearable by providing essential environmental measurements, including pollutant levels.

Building upon the details presented in Chapter 3.1.1, communication between the STM32 board and the BME680 sensor occurs via the Inter-Integrated Circuit (I2C) interface. The manufacturer's provided libraries will be utilized to facilitate this communication, ensuring seamless access to both BME680 data and BSEC data.

The project will utilize two distinct libraries for interfacing with the BME680 sensor. The BME680 library, readily available as open source on GitHub, facilitates interaction with the sensor's core functionalities like temperature, humidity, and pressure readings. Conversely, the BSEC library, provided for free by the manufacturer as a closed-source binary, unlocks advanced environmental sensing capabilities offered by the sensor. Specifically, BSEC employs higher-level signal processing to extract richer environmental data. It leverages compensated sensor values from the BME680 to provide a more comprehensive picture.

The BME680 library integration process involves incorporating its header and source files into the project directory (Figure 30). Header files, typically located in the Core/Inc folder, contain function and variable declarations that other program modules can reference. These declarations establish the interface for interacting with the library's functionalities. Conversely, source files, usually placed in the Core/Src folder, house the complete implementation of the declared functions. These files define the logic behind the library's behavior.

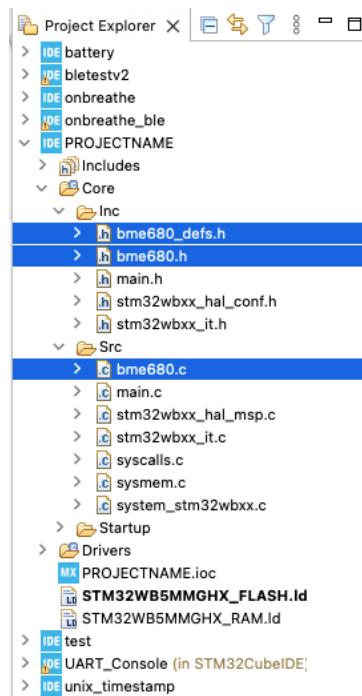


Figure 30. STM32CubeIDE project with the BME680 library.

The BSEC library needs specific linker configuration for integration. This process involves accessing the project properties through the IDE's top bar (Project -> Properties). Within the C/C++ Build settings, under the MCU GCC Linker options, a Libraries section allows specifying libraries to be linked (Figure 31). Here, the library name "algobsec" should be entered. Additionally, the library search path must be configured to point to the directory containing the "algobsec" binary. This path will vary depending on the development environment, but in this case, it corresponds to the specific GCC installation directory used for compiling the C code for the Cortex-M0+ core present in the STM32 microcontroller (as detailed in the device's datasheet).

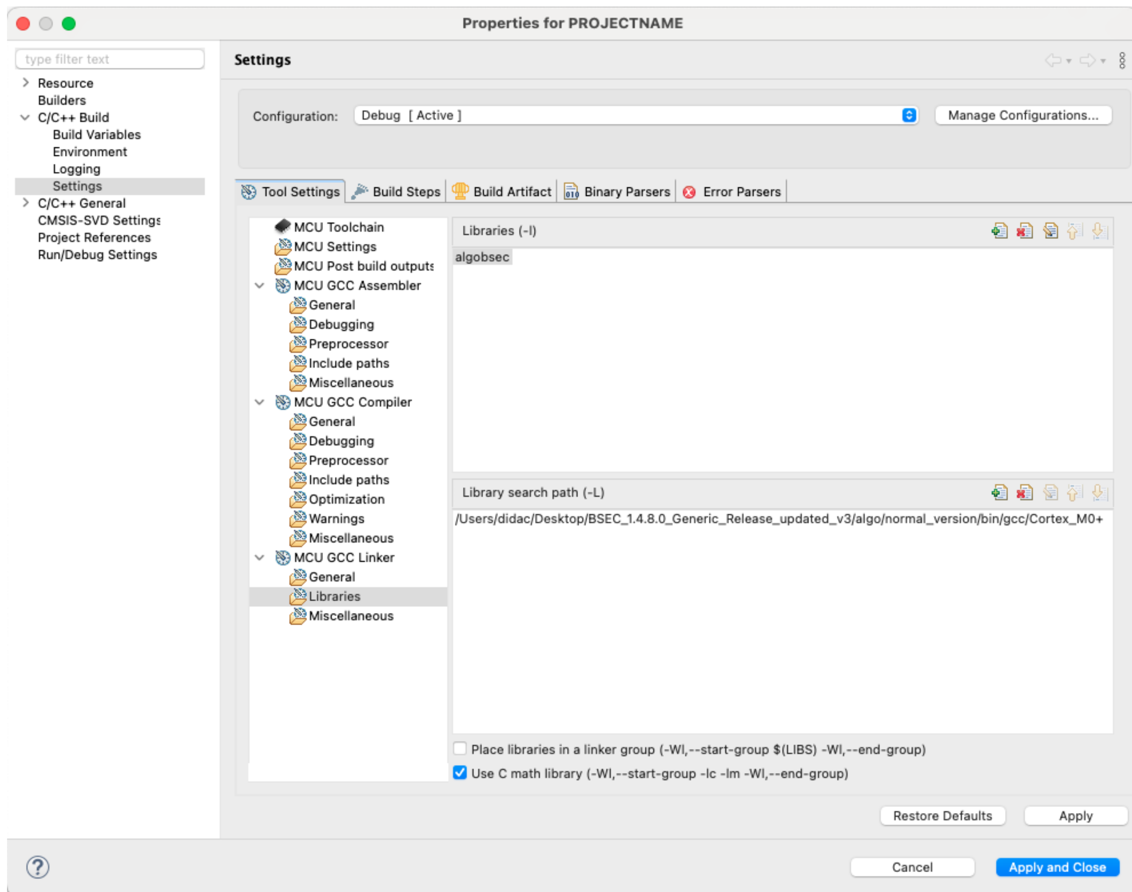


Figure 31. STM32CubeIDE project settings view.

As the BSEC library source code is closed-source, Bosch Sensortec provides a pre-compiled C class alongside three header files to enable its interaction with the BME680 sensor. This approach mirrors the integration process for the BME680 library.

Following library integration, the focus shifts to application logic development. This stage entails crafting code to interact with the BME680 sensor via I2C. To facilitate this interaction, the STM32CubeIDE's Hardware Abstraction Layer (HAL) driver provides a set of functions, including HAL_I2C_Master_Receive and HAL_I2C_Master_Transmit. These functions simplify I2C communication by allowing the programmer to specify:

- The I2C address of the BME680 sensor.
- The specific register to be read from or written to.

- The data to be sent (for write operations) or the variable to store the received data (for read operations).

This approach streamlines I2C communication, enabling the application to acquire raw sensor data (temperature, humidity, pressure) (Code 1) and subsequently pass it to the BSEC library for advanced environmental parameter processing (Code 2).

```
int8_t bme680I2cRead(uint8_t dev_id, uint8_t reg_addr, uint8_t
*reg_data,
        uint16_t len) {
    int8_t result;

    if (HAL_I2C_Master_Transmit(&hi2c1, (dev_id << 1), &reg_addr, 1,
    HAL_MAX_DELAY) != HAL_OK) {
        result = -1;
    } else if (HAL_I2C_Master_Receive(&hi2c1, (dev_id << 1) | 0x01,
reg_data,
                len, HAL_MAX_DELAY) != HAL_OK) {
        result = -1;
    } else {
        result = 0;
    }

    return result;
    }

    int8_t bme680I2cWrite(uint8_t dev_id, uint8_t reg_addr, uint8_t
    *reg_data,
        uint16_t len) {
    int8_t result;
    int8_t *buf;

    buf = malloc(len + 1);
    buf[0] = reg_addr;
    memcpy(buf + 1, reg_data, len);

    if (HAL_I2C_Master_Transmit(&hi2c1, (dev_id << 1), (uint8_t*)
buf, len + 1,
    HAL_MAX_DELAY) != HAL_OK) {
        result = -1;
    } else {
        result = 0;
    }

    free(buf);
    return result;
    }
```

Code 1. BME680 read and write functions via I2C.

```
    return_values_init ret;
    ret = bsec_iot_init(BSEC_SAMPLE_RATE_LP, 0.0f, bme680I2cWrite,
                bme680I2cRead, sleep, state_load, config_load);

    /* ... Check ret status. Is BME ok? Is BSEC ok? ... */

    /* ... Initialize variables ... */
```

```
    /* Retrieve sensor settings to be used in this time instant by
calling bsec_sensor_control */
    bsec_sensor_control(time_stamp, &sensor_settings);

    /* Trigger a measurement if necessary */
    bme680_bsec_trigger_measurement(&sensor_settings, sleep);

    /* Read data from last measurement */
    bme680_bsec_read_data(time_stamp, bsec_inputs, &num_bsec_inputs,
                sensor_settings.process_data);

    /* Time to invoke BSEC to perform the actual processing */
    bme680_bsec_process_data(bsec_inputs, num_bsec_inputs,
output_ready);

    /* ... Sample counter and state saving... */
```

Code 2. BME680 and BSEC measurements reading.

The code snippet (Code 2, line 2) exemplifies the configuration of the BSEC library's sampling rate using the BSEC_SAMPLE_RATE_LP constant. This selection prioritizes low power consumption, around 0.9 mA, while incurring a slower sampling rate of 3 seconds. The choice of sampling rate presents a trade-off between power efficiency and data acquisition frequency.

However, compiling this code initially encounters errors from the BSEC library related to VPF register arguments (Figure 32). This incompatibility arises because the BSEC library have been compiled with SoftFP, an instruction set allowing for both hardware and software floating-point operations. In contrast, STM32 projects by default utilize hardware floating-point units (FPU). To resolve this discrepancy, we need to navigate to the project properties within the IDE (Project -> Properties). Under the C/C++ Build settings and MCU Settings, locate the Floating-Point ABI option. Changing this setting from "Hardware implementation" to "Mix HW/SW implementation" instructs the compiler to accommodate the SoftFP nature of the BSEC library (Figure 33). Once this modification is implemented, the compilation errors should be rectified, enabling successful code debugging and retrieval of sensor measurements from the BME680 (Figure 34).

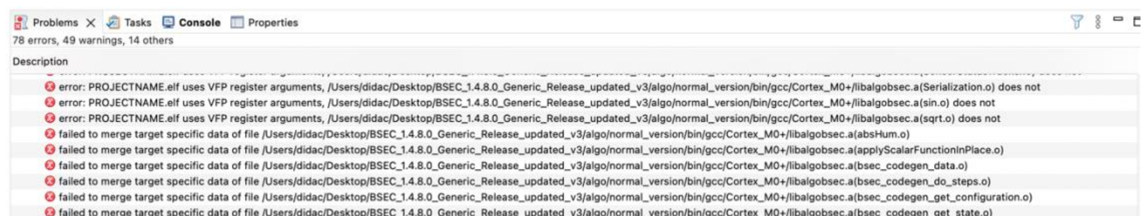


Figure 32. Errors when using hardware floating-point operations.

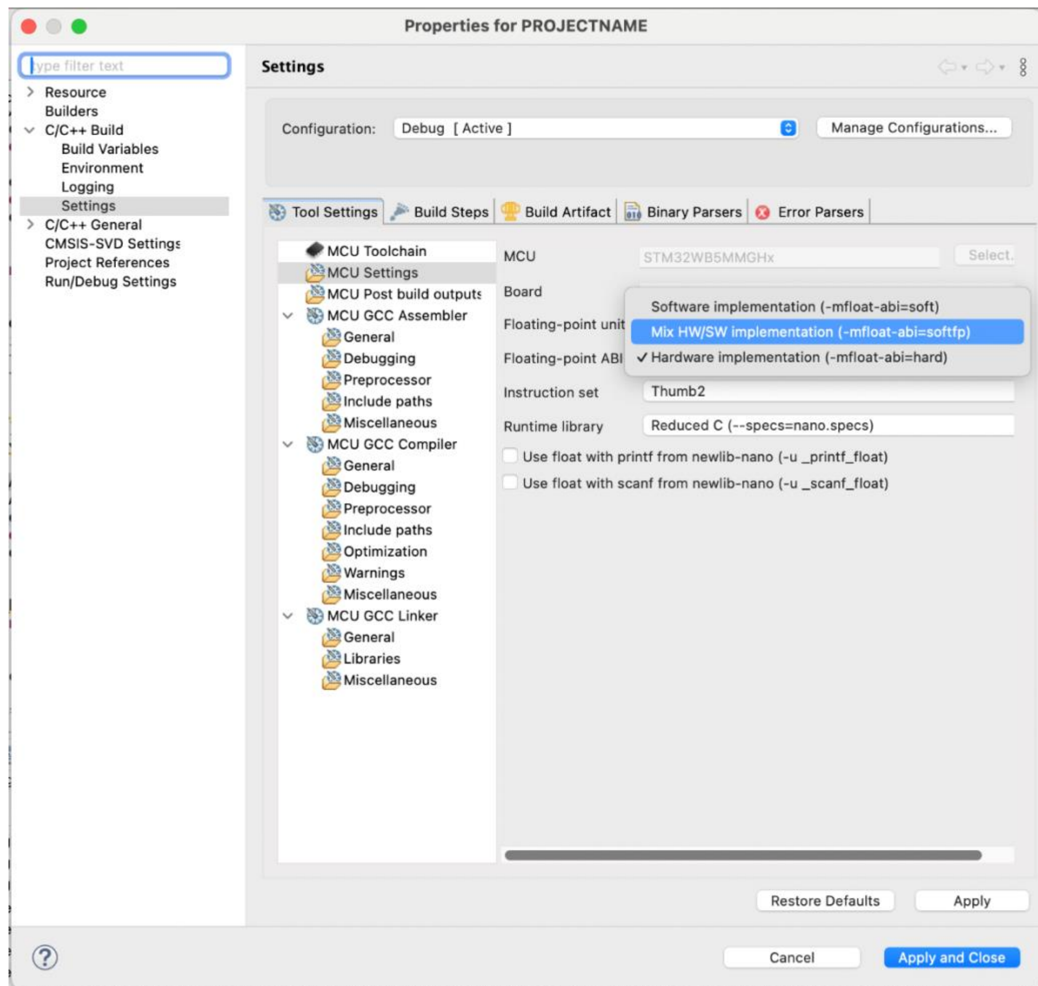


Figure 33. STM32CubeIDE project MCU Settings.

Expression	Type	Value
(x)= iaq_o	float	25
(x)= temperature_o	float	22.9400005
(x)= humidity_o	float	37.4329987
(x)= pressure_o	float	101057
(x)= co2_equivalent_o	float	0
(x)= breath_voc_equivalen	float	0
Add new expression		

Figure 34. BSEC data output.

Among the various output variables provided by the BSEC library, our focus lies on the compensated values. This selection is crucial due to potential self-heating effects introduced by components’ operation. These effects can influence the raw measurements acquired from the BME680 sensor, potentially leading to inaccuracies. The BSEC library's compensation algorithms address this challenge by factoring in these thermal influences, resulting in more reliable and representative environmental data.

3.1.3 Bluetooth Low Energy

To transmit the environmental data acquired from the BME680 sensor, the Bluetooth Low Energy (BLE) protocol must be enabled within the Device Configuration Tool. This configuration is found under Middleware and Software Packs, specifically the STM32_WPAN package (Figure 35). As the Integrated Development Environment (IDE) advises, enabling BLE needs the activation of additional peripherals crucial for its operation. These essential peripherals include the Radio Frequency (RF) module, Real-Time Clock (RTC), Reset and Clock Control (RCC), Inter-Processor Communication (IPCC), and Hardware Semaphore (HSEM).

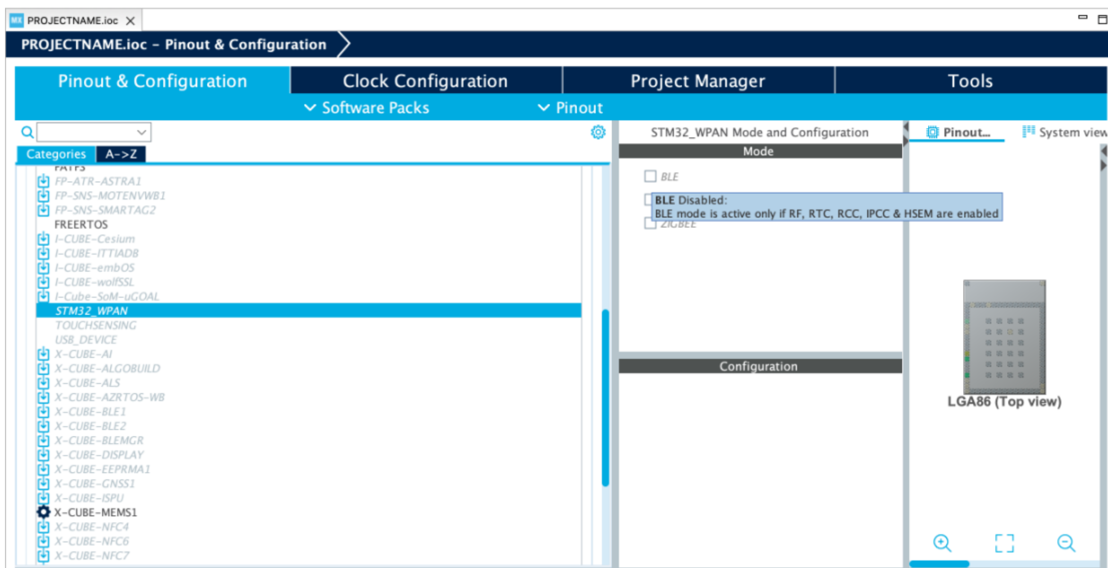


Figure 35. STM32CubeIDE STM32_WPAN package configuration.

Following the activation and configuration of the peripherals, BLE functionality can be enabled. The subsequent configuration process involves navigating to the “BLE Applications and Services” tab. Within this tab, it is necessary to modify the Server Mode setting from “Custom P2P Server” to “Custom to Custom Template”. This selection allows for the definition of custom services and characteristics specific to our application. Because of this change, two additional tabs, “BLE Advertising” and “BLE GATT”, will become available for further configuration, which will be addressed subsequently.

To configure the maximum transmission power of the embedded antenna within the microcontroller, we navigate to the “Configuration” tab. Under the “Application parameters” section, locate the parameter labelled “CFG_TX_POWER”. This parameter holds by default a value of -0.15 dBm. To achieve the maximum allowable transmission power, we modify this value to 6 dBm.

To customize the name of the wearable device as it appears during Bluetooth Low Energy (BLE) communication with the smartphone application, we navigate to the “BLE Advertising” tab. Within this tab, locate the “Advertising elements” section. Here, we enable the inclusion of the “AD_TYPE_COMPLETE_LOCAL_NAME” element by selecting “Yes”. This action will activate the corresponding field labeled “AD_TYPE_COMPLETE_LOCAL_NAME”, which currently displays the default value “XX-STM32”. We modify this value to “onBREATHE” to reflect the desired device name.

Once these fundamental BLE configuration steps are completed, we can proceed to GATT (Generic Attribute Profile) configuration. GATT defines the services and characteristics that facilitate data exchange between BLE devices. To define the data structure for our application, we will navigate to the "BLE GATT" tab. This tab prompts the user to specify the number of services required. Since each service can only accommodate a maximum of five characteristics, we will establish two services to transmit all the necessary environmental data points (Table 6).

Service	Characteristic	UUID	Variable	Unit
AirService (UUID: 00)	AirTemp	00	Sensor-compensated temperature	° C
	AirPres	01	Raw pressure	Pa
	AirHum	10	Sensor-compensated relative humidity	%
	AirVOCs	11	Breath-VOC equivalents	ppm
	AirCO2	100	CO2 equivalents	ppm
AirService2 (UUID: 01)	IAQAcc	00	IAQ accuracy status	-
	AirIAQ	01	Index for Air Quality	-
	Batt	10	Battery level from LTC2942-1	%

Table 6. GATT services and characteristics for data transmission.

Since the data transmitted via these characteristics is intended for use by the mobile application, the "readable" property will be enabled for each characteristic.

To facilitate the background update of characteristics with the latest environmental data, a dedicated task needs to be implemented (Figure 36). This involves first declaring the task within Core/Inc/app_conf.h specifying its properties. Next, in STM32_WPAN/App/app_ble.c, the task is registered and set. Finally, the core functionality of the task, responsible for periodically reading the environmental data and updating the characteristic values, is implemented within STM32_WPAN/App/custom_app.c.

Figure 36. STM32CubeIDE task configuration.

The custom_app.c file implements the core functionality of the background task responsible for updating BLE characteristics. Within this function, the task retrieves the latest environmental data from the BME680 sensor. This retrieved data is then used to update the corresponding characteristics within the BLE service (Code 3), ensuring the mobile application receives the most recent sensor readings.

```
char temp[4];
sprintf(temp, "%2.2f", output.temperature);
Custom_STM_App_Update_Char(CUSTOM_STM_AIRTEMP, &temp);

/* ... Same with the other parameters ... */
```

Code 3. BLE characteristic update.

However, to enable BLE advertising functionality, the factory-installed firmware on the STM32WB microcontroller lacks the necessary wireless stack. This essential software component can be installed and activated using a dedicated tool called STM32CubeProgrammer. While STM32CubeProgrammer offers various functionalities like memory editing, programming, and serial communication visualization, the feature of interest for this purpose is the “firmware upgrade services”. This specific functionality within STM32CubeProgrammer facilitates the upload and installation of the missing wireless stack onto the microcontroller.

Within STM32CubeProgrammer, the "firmware upgrade services" screen provides valuable information regarding the microcontroller’s firmware. By clicking "Read FUS infos," we can verify the current stack version. As the displayed version is 0.0.0.0, it confirms that the wireless stack is not yet installed (Figure 37 a). Following an upload of the BLE stack using the firmware file, the stack version field will display the correct version number (Figure 37 b). Finally, to activate the uploaded wireless stack and enable BLE advertising functionality, the "Start Wireless Stack" button is clicked.

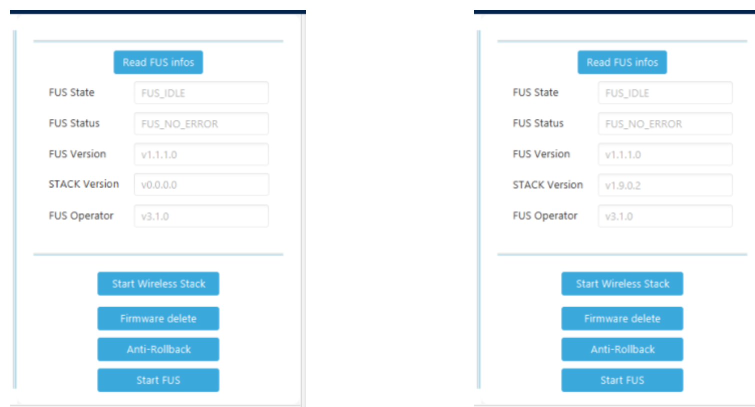


Figure 37. a) FUS information before wireless stack installation. b) FUS information after wireless stack installation.

To verify successful BLE advertising functionality, a commercially available application such as “BLE Scanner (Connect & Notify)” from the Google Play Store can be employed. This application allows users to scan for nearby BLE devices and read the characteristic values transmitted by our wearable device (Figure 38).

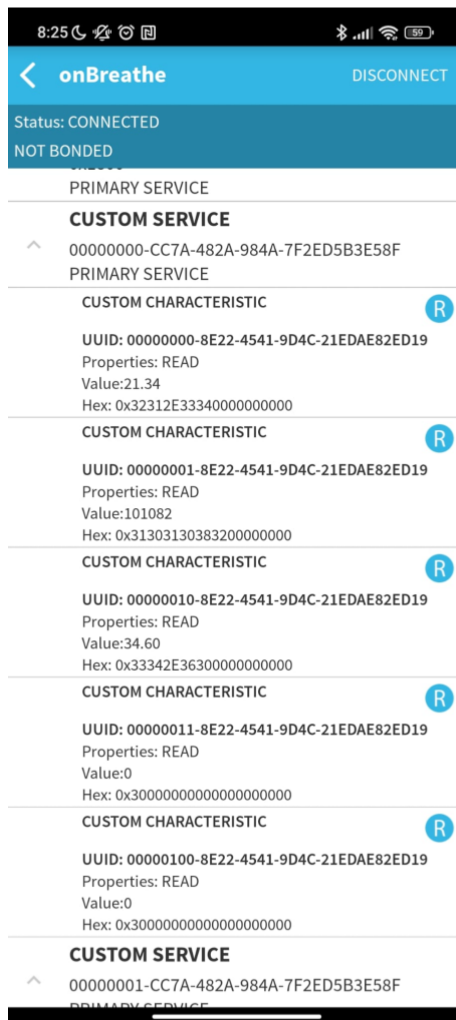


Figure 38. Reading BLE characteristics from the wearable with a commercial app.

3.1.4 Battery gauge (LTC2942-1)

Analogous to the communication established with the BME680 sensor, the interaction between the LTC2942-1 battery gauge and the microcontroller leverages the I2C protocol.

Within the project's main code, configuration of the LTC2942-1 battery gauge involves setting the control register for automatic operation (Code 4), designating the alert pin as an input due to its connection to the battery charger, and defining the maximum electric charge (mAh) of the specific battery (in this case, 600 mAh). This latter value is crucial for two reasons: calculating the optimal coulomb counter prescaler for accurate capacity readings and determining the battery's remaining charge percentage.

```
	set_control_reg_value(&hi2c1, 0xFA); // FA = 11111010, automatic
and STAT input
	set_battery_full_mah(&hi2c1, 600); // set max value of 600 mAh
(battery)

void set_battery_full_mah(I2C_HandleTypeDef *hi2c, uint16_t mAh) {
	uint8_t i = 0;
```

```
        float qLSB = 0, temp = mAh;

        if (mAh > maxAmh) {
                return;
        }

        qLSB = temp / 65536;

        uint8_t prescalarTable[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };

        for (i = 0; i < 8; i++) {
                temp = (q_LSB * prescalarTable[i]) / (128);
                if (qLSB <= temp) {
                        break;
                }
        }

        if (i >= 8) {
                return;
        }

        maxAmh = mAh;

        prescalar = prescalarTable[i];

        qLSB = (q_LSB * prescalar) / (128);

        temp = mAh;
        temp = temp / qLSB;
        mAh = (uint16_t) temp;
}
```

Code 4. Battery gauge configuration lines.

Leveraging the previously configured LTC2942-1 gauge, the accumulated charge registers can now be accessed to determine the battery's charge percentage (Code 5). This calculated value can then be written to a corresponding BLE characteristic, following a similar process outlined in Chapter 3.1.3.

```
const float q_LSB = 0.085; // mAh


float get_mAh(uint8_t *acc_charge_reg_buf) {
        uint16_t data = (acc_charge_reg_buf[1] | (acc_charge_reg_buf[0]
<< 8)); // 0 is MSB and 1 is LSB
        float mAh = (float) (data * q_LSB * prescalar) / (128);

        return mAh;
}


float get_battery_percentage(float mAh) {
        return (mAh * 100 / maxAmh);
}
```

Code 5. Battery gauge battery percentage functions.

4 PCB Miniaturization

This chapter addresses the miniaturization of the existing PCB layout. The initial PCB design was provided in Eagle software format. To maintain consistency and efficiency, the miniaturization process will also be conducted within the Eagle environment.

4.1 Eagle

Eagle is a widely used electronic design automation (EDA) tool that allows for schematic capture, printed circuit board (PCB) layout design, and integrated functionalities for automating tasks like component placement and routing. While Eagle is commercially licensed software, students can benefit from a free one-year educational license, making it a cost-effective choice for the project.

Eagle provides a user-friendly interface with two primary workspaces: schematic and board design. The schematic editor allows for symbol placement and connection definition, visually representing the components and their electrical connectivity on the PCB. The board design workspace facilitates the physical layout of these components, where their placement and routing are optimized for size and functionality.

The miniaturization process will primarily use the Eagle board design window. While the schematic editor will be used to remove the UART component, which is no longer needed, this change will automatically propagate to the board layout within the board design window.

The miniaturization process will start with a strategic via removal. Vias will be eliminated where possible to minimize signal path length and improve overall board density. Next, component placement will be optimized through a process of strategically moving components closer together while maintaining adequate spacing to prevent electrical shorts and ensuring manufacturability. This will prioritize minimizing right-angle traces, as these can introduce signal integrity issues. Following component optimization, strategically placed vias will be reintroduced to maintain electrical connectivity as needed. The 'GND' command within Eagle will be utilized to efficiently establish a continuous ground connection between the top and bottom ground planes. Finally, with the layout optimized, the PCB dimensions will be reduced to achieve the desired miniaturized form factor.

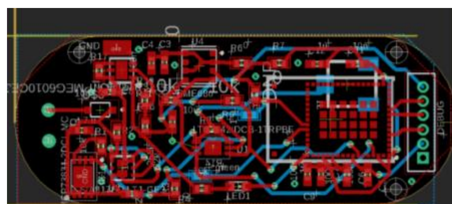


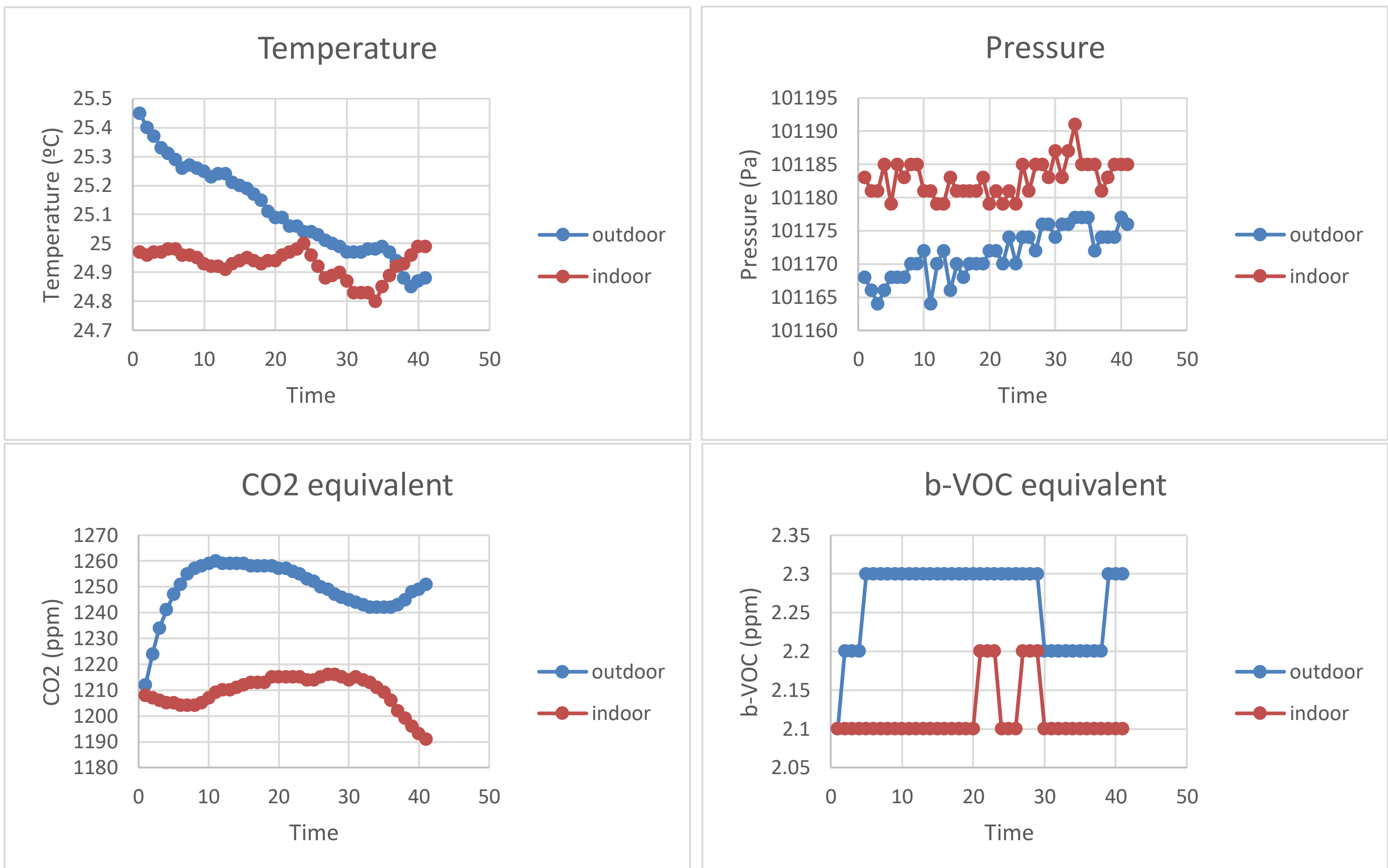
Figure 39. Board view of the miniaturized PCB.

The miniaturization process achieved a significant size reduction for the wearable PCB. The final dimensions are 39.30 mm x 15.10 mm, representing a decrease of approximately 20% in width compared to the initial design.

5 Validation and results

This chapter focuses on initial functionality testing of the printed circuit board (PCB). This includes verifying the performance of the BME680 sensor and the data transmission capabilities of the STM32 microcontroller via Bluetooth Low Energy (BLE). Due to the unavailability of the mobile application being developed by our collaborator onLean, a temporary solution was implemented using App Inventor. This temporary app facilitates storage by reading sensor readings, transmitting them to Thingspeak, and allowing for their download in CSV format.

To calibrate the BME680 sensor according to the manufacturer's datasheet, we exposed the sensor's PCB to indoor and outdoor environments for 30 minutes each. During this period, the sensor's IAQ accuracy moved from 0 (calibrating) to 1 (low accuracy). Despite exposing the sensor to distinct air qualities, we were unable to achieve higher IAQ accuracy levels of 2 (medium accuracy) or 3 (high accuracy). This limitation is significant as the b-VOC and CO_2 equivalents are estimated based on IAQ levels. Achieving level 1 accuracy suggests the need to expose the sensor to both high-quality air (e.g., outdoor air) and low-quality air (e.g., a confined space with exhaled breath), which was not feasible due to resource constraints. Following the calibration, we proceeded with measurements by exposing the PCB to both indoor and outdoor environments for an additional 10 minutes (Figure 40).



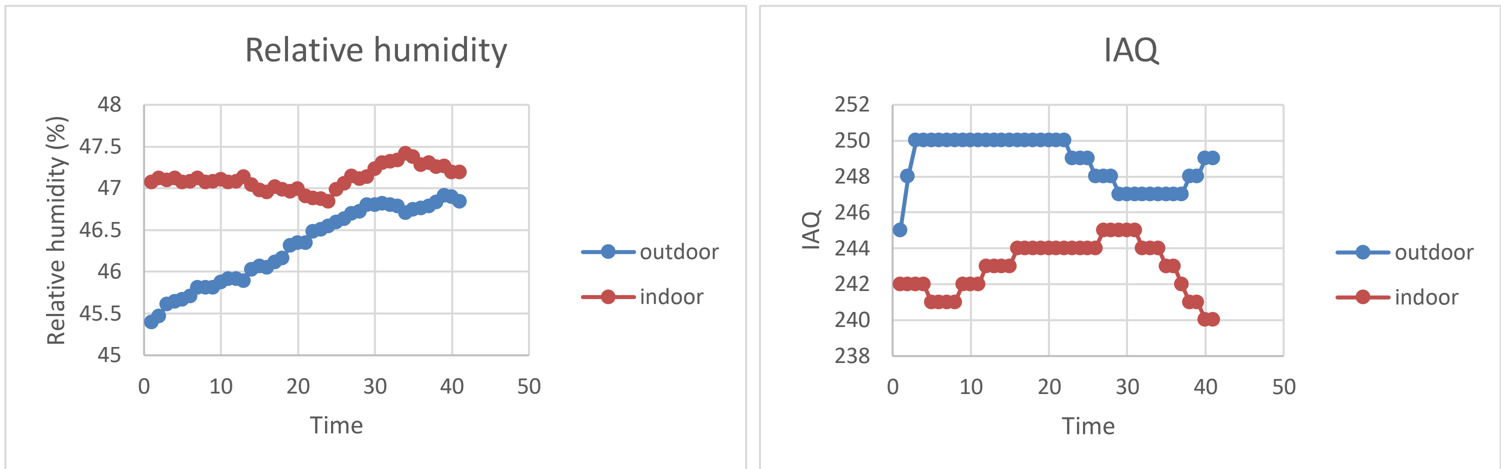


Figure 40. PCB environmental measurements after BME680's calibration.

The temperature data, as shown in the first graph, indicates that the outdoor temperature started higher and gradually decreased from approximately 25.4 °C to around 24.7 °C over the measurement period. In contrast, the indoor temperature remained relatively stable, fluctuating slightly around 24.9°C. This stability in indoor temperature suggests effective environmental control within the indoor space.

The pressure readings, presented in the second graph, reveal a notable difference between indoor and outdoor environments. The outdoor pressure showed a gradual increase from around 101,165 Pa to about 101,175 Pa. Indoor pressure, however, remained consistently higher, fluctuating narrowly around 101,185 Pa. The stability of the indoor pressure compared to the more variable outdoor readings could be attributed to the sealed nature of the indoor environment.

The relative humidity data depicted in the third graph shows that outdoor humidity increased from approximately 46.5% to 47.5%, indicating a rising trend in moisture content. Indoor humidity levels, on the other hand, remained more constant, ranging narrowly between 46.5% and 47.0%.

The b-VOC equivalent graph reveals distinct patterns for indoor and outdoor environments. Outdoor b-VOC levels remained higher and fluctuated between 2.1 ppm and 2.3 ppm. Indoor b-VOC levels were consistently lower and more stable, remaining around 2.1 ppm. The lower and stable indoor b-VOC levels indicate better air quality in terms of volatile organic compounds.

The CO_2 equivalent levels, as shown in the fifth graph, indicate that outdoor CO_2 levels started around 1210 ppm, peaked at approximately 1270 ppm, and then decreased to around 1240 ppm. Indoor CO_2 levels, however, remained relatively stable, ranging from about 1180 ppm to 1210 ppm. The lower and more stable indoor CO_2 levels could be attributed to good ventilation and air quality control within the indoor environment.

The IAQ graph illustrates that indoor air quality remained better throughout the measurement period. The indoor IAQ index fluctuated slightly around 240, while the outdoor IAQ index started around 248 and showed a slight decreasing trend. The lower IAQ index indoors indicates a higher quality of indoor air compared to outdoor air.

6 Conclusions and future perspectives

The project aimed to address the limitations of traditional air quality monitoring stations by creating a device that tracks an individual's exposure to VOCs throughout the day. We successfully achieved the project's initial goals, demonstrating the feasibility of the core concept.

Firstly, we gained a comprehensive understanding of VOCs, which was crucial for selecting appropriate sensor technology (Goal 1). Secondly, we evaluated the feasibility of using Internet of Things (IoT) and Bluetooth Low Energy (BLE) technologies for efficient data transmission. This evaluation ensured the device's compatibility with existing communication protocols and minimized power consumption (Goal 2). Thirdly, we successfully implemented the Inter-Integrated Circuit (I2C) communication protocol for seamless communication between the microcontroller and onboard components (Goal 3). Fourthly, we studied the selected components, including the STM32 microcontroller, NCP705MT33 voltage regulator, BME680 environmental sensor, LTC2942-1 battery gauge, and MCP73831 battery charger, for optimal performance and power efficiency (Goal 4). This in-depth analysis ensured the device's functionality while minimizing its energy footprint. Fifthly, we developed robust and efficient software for the microcontroller to manage sensor data acquisition, processing, and communication with minimal power consumption (Goal 5). Finally, we successfully miniaturized the PCB design almost a 20% while maintaining functionality and reliability (Goal 6). This miniaturization is crucial for creating a comfortable and wearable device.

The initial functionality testing delivered promising results. The BME680 sensor successfully collected data on temperature, pressure, humidity, CO_2, IAQ and b-VOC equivalents. We observed clear distinctions between indoor and outdoor readings for these parameters. For instance, outdoor temperature readings showed a gradual decrease, while indoor temperatures remained relatively stable. Similarly, outdoor pressure readings exhibited a slight increase, whereas indoor pressure remained more consistent. These variations suggest the device's potential to capture the dynamic nature of air quality throughout the day.

However, achieving level 1 accuracy on the IAQ sensor calibration presented a challenge (Goal 7). This limitation restricts the accuracy of b-VOC and CO_2 equivalent readings. The manufacturer’s datasheet specifies that higher IAQ accuracy levels (2 for medium and 3 for high) are achieved by exposing the sensor to environments with both high and low air quality. Due to resource constraints, we were unable to replicate these conditions, resulting in a lower level of accuracy.

Overall, the project successfully developed a functional prototype for a low-power wearable air quality monitor. The initial testing demonstrates the device’s ability to collect and transmit environmental data. Future work should focus on several key areas. Firstly, refining the calibration process to achieve higher IAQ accuracy levels. This will improve the accuracy of b-VOC and CO_2 equivalent readings, providing users with more reliable information about their exposure to pollutants. Secondly, conducting long-term wearability studies will assess user comfort and compliance. A comfortable and user-friendly design is essential for encouraging individuals to wear the device consistently and maximize data collection. Thirdly, integrating additional sensors to capture a broader

range of air pollutants would provide a more comprehensive picture of an individual's environmental exposure. Finally, validating the device's performance in real-world settings with diverse populations. This validation will ensure the device's effectiveness in various environments and for different user demographics.

By addressing these aspects, the project can progress towards a robust and user-friendly wearable air quality monitor that empowers individuals to track their exposure to pollutants and make informed decisions about their health and environment.

Budget

The budget was generated by Eurocircuits, a manufacturer that provides PCB services.

Bill of Materials (BOM):

Reference designators	Quantity	MPN	Manufacturer	Description	Price
C1, C2	2	GPC0603475-10	Generic part	4.7μF ±10% 10V Ceramic Capacitor X5R 0603 (1608 Metric)	Free
C3, C4, C5, C6, C9	5	GPC0402104	Generic part	0.1μF ±10% 50V Ceramic Capacitor X7R 0402 (1005 Metric)	Free
C7, C12	2	GPC0402105-16	Generic part	1μF ±10% 16V Ceramic Capacitor X5R 0402 (1005 Metric)	Free
C8, C10	2	GPC0402106-6.3	Generic part	10μF ±20% 6.3V Ceramic Capacitor X5R 0402 (1005 Metric)	€ 0.36
C11	1	GPC0402103-25	Generic part	10000pF ±10% 25V Ceramic Capacitor X7R 0402 (1005 Metric)	Free
R1, R4, R9, R11	4	GPR040210K	Generic part	0402, Res 10.0kOhm, 50V, 1.0%, 62.5mW	Free
R2	1	GPR0402100K	Generic part	0402, Res 100.0kOhm, 50V, 1.0%, 62.5mW	Free
R3, R6, R7, R17	4	GPR04020R	Generic part	0 Ohms Jumper 0.1W, 1/10W Chip Resistor 0402 (1005 Metric) Automotive AEC-Q200 Thick Film	Free
R5, R13	2	GPR04021K	Generic part	0402, Res 1.0kOhm, 50V, 1.0%, 62.5mW	Free
LED1	1	598-8081-107F	Dialight	LED GREEN CLEAR 0603 SMD	€ 1.35
DEBUG	1	5-104071-8	TE Connectivity AMP Connectors	Connector Header Through Hole 6 position 0.050" (1.27mm)	€ 1.96
R8	1	GPR04021R	Generic part	0402, Res 1.0Ohm, 50V, 1.0%, 62.5mW	Free
R10	1	GPR040282K	Generic part	0402, Res 82.0kOhm, 50V, 1.0%, 62.5mW	Free
IC1	1	MCP73831-2DCI/MC	Microchip Technology	Charger IC Lithium Ion/Polymer 8-DFN (2x3)	€ 1.71
IC4	1	NCP705MTADJTCG	ON Semiconductor	Linear Voltage Regulator IC Positive Adjustable 1 Output 500mA 6-WDFN (2x2)	€ 1.48

J1	1	SM02B-SRSS-TB (LF)(SN)	JST Sales America Inc.	2 Positions Header Connector 0.039" (1.00mm) Surface Mount, Right Angle Tin	€ 1.21
R16	1	ERJ-2RKF2553X	Panasonic Electronic Components	255 kOhms ±1% 0.1W, 1/10W Chip Resistor 0402 (1005 Metric) Automotive AEC-Q200 Thick Film	€ 1
U$1, U$2	2	1548-0-57-15-00-00-03-0	Mill-Max Manufacturing Corp.	Contact Spring Surface Mount	€ 1.99
U$3	1	PMEG6010CEJF	Nexperia	Diode Schottky 60V 1A (DC) Surface Mount SC-90	€ 1
U1	1	LTC2942CDCB#TRMPBF	Linear Technology/Analog Devices	Battery Battery Monitor IC Lithium-Ion 6-DFN (2x3)	€ 7.52
U2	1	SIA817EDJ-T1-GE3	Vishay Siliconix	P-Channel 30V 4.5A (Tc) 1.9W (Ta), 6.5W (Tc) Surface Mount PowerPAK® SC-70-6 Dual	€ 1
U3	1	STM32WB5MMGH6TR	STMicroelectronics	802.15.4, Bluetooth Bluetooth v5.0, Thread, Zigbee® Transceiver Module 2.4GHz Integrated, Chip Surface Mount	€ 23.94
U4	1	BME680	Bosch	Gas, Humidity, Pressure, Temperature Sensor I²C Output	€ 21.61

Production costs for printed circuit boards (PCBs) are directly tied to the volume ordered. Higher quantities allow manufacturers to spread setup costs and material overhead across a larger number of units, resulting in a lower per-board price (Table 7).

Quantity	1	3
PCB Services	€ 75.85	€ 109.23
Components	€ 66.15	€ 130.50
Assembly	€ 390.42	€ 410.49
Express transport	€ 7.36	
Total	€ 532.42	€ 593.23

Table 7. PCB production costs.

References

[1] World Health Organization: WHO, “Air pollution.” Accessed: Apr. 05, 2024. [Online]. Available: https://www.who.int/health-topics/air-pollution

[2] World Health Organization: WHO, “Ambient (outdoor) air pollution.” Accessed: Apr. 05, 2024. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/ambient-(outdoor)-air-quality-and-health

[3] European Environment Agency, “Health impacts of air pollution in Europe, 2022.” Accessed: Apr. 05, 2024. [Online]. Available: https://www.eea.europa.eu/publications/air-quality-in-europe-2022/health-impacts-of-air-pollution

[4] American Lung Association, “Volatile Organic Compounds.” Accessed: Apr. 07, 2024. [Online]. Available: https://www.lung.org/clean-air/indoor-air/indoor-air-pollutants/volatile-organic-compounds

[5] US EPA, “What are volatile organic compounds (VOCs)?” Accessed: Apr. 07, 2024. [Online]. Available: https://www.epa.gov/indoor-air-quality-iaq/what-are-volatile-organic-compounds-vocs

[6] US EPA, “What are SVOCs (and VOCs)?” Accessed: Apr. 07, 2024. [Online]. Available: https://www.epa.gov/east-palestine-oh-train-derailment/what-are-svocs-and-vocs

[7] E. David and V.-C. Niculescu, “Volatile Organic Compounds (VOCs) as Environmental Pollutants: Occurrence and Mitigation Using Nanomaterials,” *Int J Environ Res Public Health*, vol. 18, no. 24, p. 13147, Dec. 2021, doi: 10.3390/ijerph182413147.

[8] J. Manyika *et al.*, “The Internet of Things: Mapping the value beyond the hype,” *McKinsey Global Institute*, Jun. 2015.

[9] IndustryARC, “Bluetooth Smart/Bluetooth Low Energy Market by Applications 2020.” Accessed: Apr. 10, 2024. [Online]. Available: https://www.industryarc.com/PressRelease/43/bluetooth-smart-lowenergy.html

[10] M. Bhargava, *IoT Projects with Bluetooth Low Energy*. Packt Publishing, 2017.

[11] J. Catsoulis, *Designing Embedded Hardware*. O’Reilly, 2002.

[12] STMicroelectronics, “STM32WB5MMG Datasheet.” Feb. 26, 2024. Accessed: Apr. 16, 2024. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32wb5mmg.pdf

[13] STMicroelectronics, “Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5.4 and 802.15.4 radio solution.” Aug. 23, 2023. Accessed: Apr. 16, 2024. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32wb55cc.pdf

[14] Semiconductor Components Industries, “NCP705 LDO Regulator - Ultra-Low Quiescent Current, IQ 13 A, Ultra-Low Noise.” Sep. 2019. Accessed: Apr. 20, 2024. [Online]. Available: https://www.onsemi.com/pdf/datasheet/ncp705-d.pdf

[15] Bosch Sensortec GmbH, “BME680. Low power gas, pressure, temperature & humidity sensor.” Feb. 2024. Accessed: Apr. 24, 2024. [Online]. Available: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme680-ds001.pdf

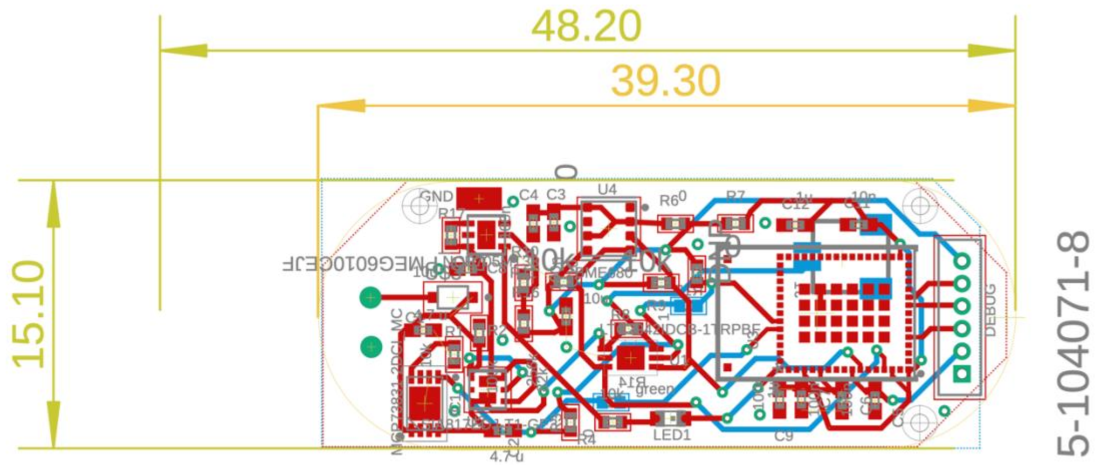
[16] Linear Technology Corporation, “LTC2942-1. 1A Battery Gas Gauge with Internal Sense Resistor and Temperature/Voltage Measurement.” Accessed: Apr. 28, 2024. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/29421f.pdf

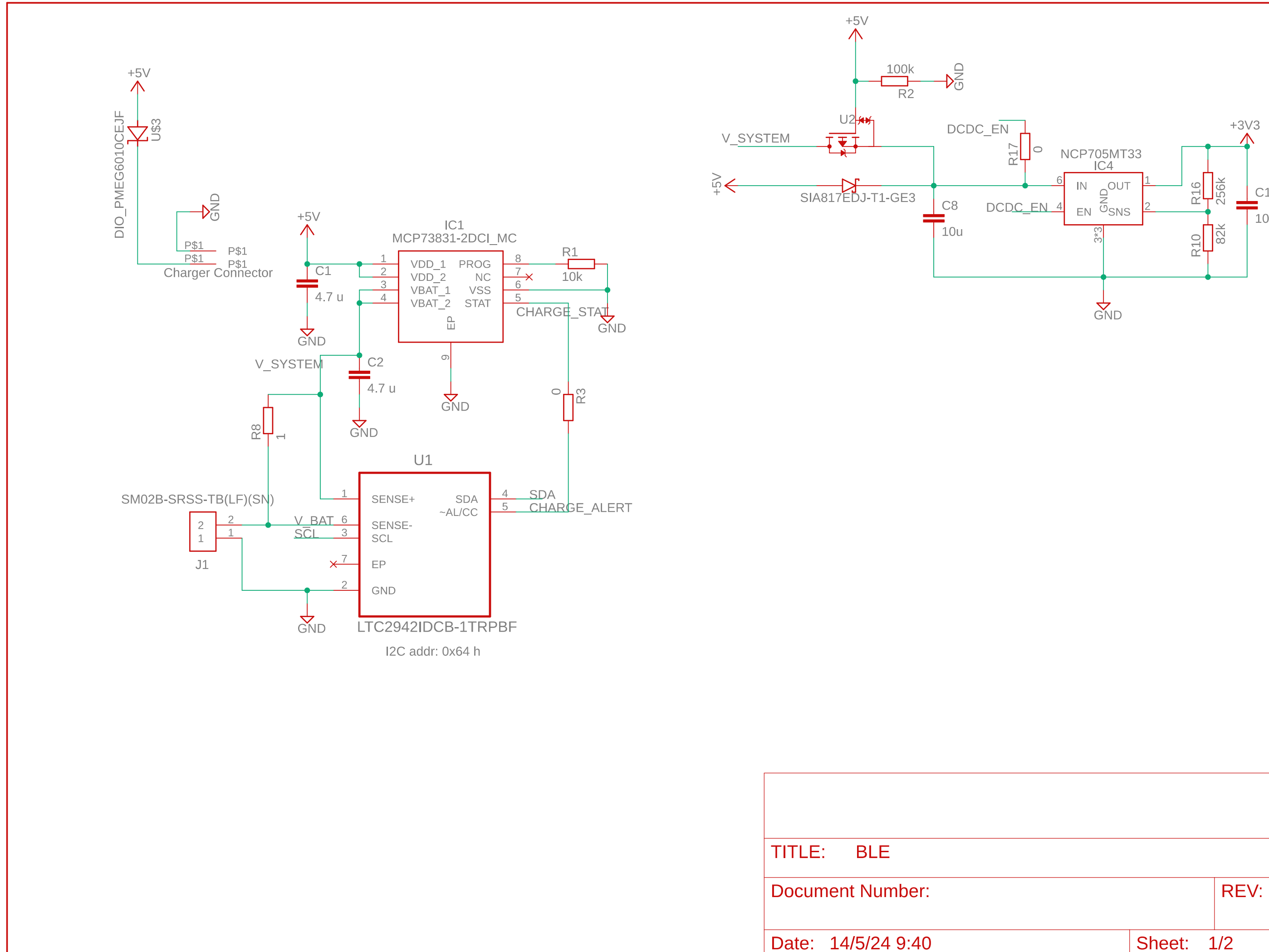
[17] Microchip Technology Inc., “MCP73831/2. Miniature Single-Cell, Fully Integrated Li-Ion, Li-Polymer Charge Management Controllers,” Feb. 2020, Accessed: Apr. 28, 2024. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP73831-Family-Data-Sheet-DS20001984H.pdf

[18] STMicroelectronics, “STM32 hardware debugging & programming tools - Discover the STLINK portfolio.” Mar. 24, 2023. Accessed: May 08, 2024. [Online]. Available: https://www.st.com/resource/en/product_presentation/stlink-debugging-and-programming-tools-overview.pdf

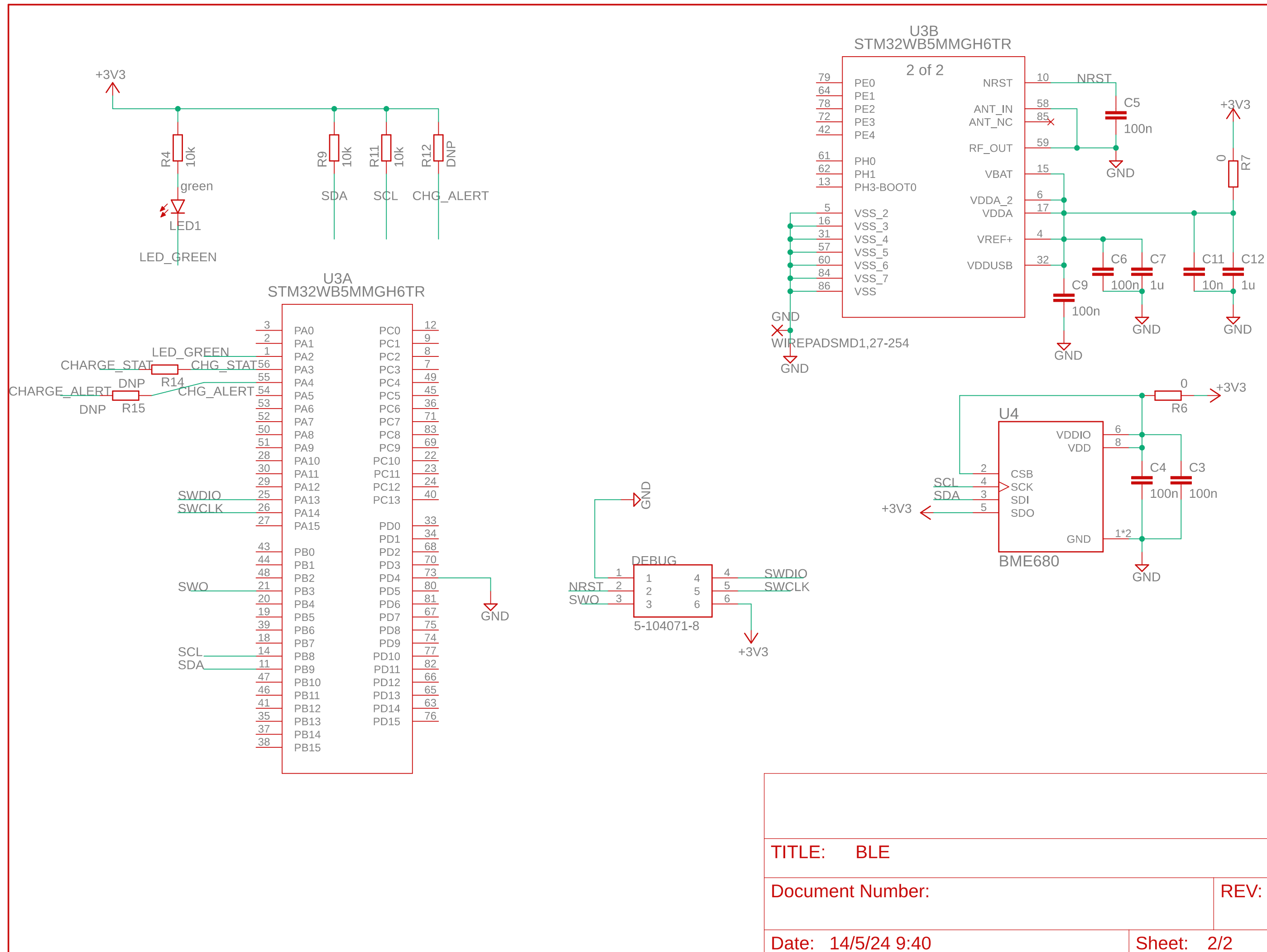
Appendices

Appendix I – PCB





29/5/24 13:08 /Users/didac/Library/CloudStorage/OneDrive-URV/OnBreathe/PCB/BLEv2/BLE.sch (Sheet: 1/2)



29/5/24 13:08 /Users/didac/Library/CloudStorage/OneDrive-URV/OnBreathe/PCB/BLEv2/BLE.sch (Sheet: 2/2)

Appendix II – STM32 code

main.c

```
/* USER CODE BEGIN Header */
/**

**********************************************************************
********
 * @file           : main.c
 * @brief          : Main program body

**********************************************************************
********
 * @attention
 *
 * Copyright (c) 2024 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the
LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *

**********************************************************************
********
 */
/* USER CODE END Header */
/* Includes ----------------------------------------------------------
--------*/
#include "main.h"

/* Private includes --------------------------------------------------
--------*/
/* USER CODE BEGIN Includes */
#include "bsec_integration.h"
#include "LTC2942.h"
#include "app_common.h"
#include "dbg_trace.h"
#include "ble.h"
#include "custom_app.h"
#include "custom_stm.h"
#include "stm32_seq.h"
/* USER CODE END Includes */

/* Private typedef ---------------------------------------------------
--------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ----------------------------------------------------
--------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -----------------------------------------------------
--------*/
```

```
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ------------------------------------------------
---------*/
I2C_HandleTypeDef hi2c1;

IPCC_HandleTypeDef hipcc;

RTC_HandleTypeDef hrtc;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes ----------------------------------------
---------*/
void SystemClock_Config(void);
void PeriphCommonClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_IPCC_Init(void);
static void MX_RTC_Init(void);
static void MX_I2C1_Init(void);
static void MX_RF_Init(void);
/* USER CODE BEGIN PFP */
int8_t bme680I2cRead(uint8_t dev_id, uint8_t reg_addr, uint8_t
*reg_data,
            uint16_t len);
int8_t bme680I2cWrite(uint8_t dev_id, uint8_t reg_addr, uint8_t
*reg_data,
            uint16_t len);
uint32_t state_load(uint8_t *state_buffer, uint32_t n_buffer);
uint32_t config_load(uint8_t *config_buffer, uint32_t n_buffer);
void sleep(uint32_t t_ms);
/* USER CODE END PFP */

/* Private user code -------------------------------------------------
---------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void) {

      /* USER CODE BEGIN 1 */

      /* USER CODE END 1 */

      /* MCU Configuration---------------------------------------------
------------*/

      /* Reset of all peripherals, Initializes the Flash interface and
the Systick. */
      HAL_Init();
      /* Config code for STM32_WPAN (HSE Tuning must be done before
system clock configuration) */
```

```
	MX_APPE_Config();

	/* USER CODE BEGIN Init */

	/* USER CODE END Init */

	/* Configure the system clock */
	SystemClock_Config();

	/* Configure the peripherals common clocks */
	PeriphCommonClock_Config();

	/* IPCC initialisation */
	MX_IPCC_Init();

	/* USER CODE BEGIN SysInit */

	/* USER CODE END SysInit */

	/* Initialize all configured peripherals */
	MX_GPIO_Init();
	MX_RTC_Init();
	MX_I2C1_Init();
	MX_RF_Init();
	/* USER CODE BEGIN 2 */
	return_values_init ret;
	ret = bsec_iot_init(BSEC_SAMPLE_RATE_LP, 0.0f, bme680I2cWrite,
			bme680I2cRead, sleep, state_load, config_load);

	if (ret.bme680_status) {
		/* Could not intialize BME680 */
		return (int) ret.bme680_status;
	} else if (ret.bsec_status) {
		/* Could not intialize BSEC library */
		return (int) ret.bsec_status;
	}
	set_control_reg_value(&hi2c1, 0xFA); // FA = 11111010, automatic
and STAT input
	set_battery_full_mah(&hi2c1, 600); // set max value of 600 mAh
(battery)

	/* USER CODE END 2 */

	/* Init code for STM32_WPAN */
	MX_APPE_Init();

	/* Infinite loop */
	/* USER CODE BEGIN WHILE */
	while (1) {
		/* USER CODE END WHILE */
		MX_APPE_Process();
		/* USER CODE BEGIN 3 */
	}
	/* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void) {
```

```
	RCC_OscInitTypeDef RCC_OscInitStruct = { 0 };
	RCC_ClkInitTypeDef RCC_ClkInitStruct = { 0 };

	/** Configure the main internal regulator output voltage
	 */
	__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

	/** Initializes the RCC Oscillators according to the specified
parameters
	 * in the RCC_OscInitTypeDef structure.
	 */
	RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI
			| RCC_OSCILLATORTYPE_LSI1 | RCC_OSCILLATORTYPE_HSE
			| RCC_OSCILLATORTYPE_MSI;
	RCC_OscInitStruct.HSEState = RCC_HSE_ON;
	RCC_OscInitStruct.HSIState = RCC_HSI_ON;
	RCC_OscInitStruct.MSIState = RCC_MSI_ON;
	RCC_OscInitStruct.HSICalibrationValue =
RCC_HSICALIBRATION_DEFAULT;
	RCC_OscInitStruct.MSICalibrationValue =
RCC_MSICALIBRATION_DEFAULT;
	RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_10;
	RCC_OscInitStruct.LSIState = RCC_LSI_ON;
	RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
	if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
		Error_Handler();
	}

	/** Configure the SYSCLKSource, HCLK, PCLK1 and PCLK2 clocks
dividers
	 */
	RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK4 |
RCC_CLOCKTYPE_HCLK2
			| RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
RCC_CLOCKTYPE_PCLK1
			| RCC_CLOCKTYPE_PCLK2;
	RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
	RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
	RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
	RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
	RCC_ClkInitStruct.AHBCLK2Divider = RCC_SYSCLK_DIV1;
	RCC_ClkInitStruct.AHBCLK4Divider = RCC_SYSCLK_DIV1;

	if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) !=
HAL_OK) {
		Error_Handler();
	}
}

/**
 * @brief Peripherals Common Clock Configuration
 * @retval None
 */
void PeriphCommonClock_Config(void) {
	RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = { 0 };

	/** Initializes the peripherals clock
	 */
	PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_SMPS
			| RCC_PERIPHCLK_RFWAKEUP;
```

```
	PeriphClkInitStruct.RFWakeUpClockSelection =
RCC_RFWKPCLKSOURCE_HSE_DIV1024;
	PeriphClkInitStruct.SmpsClockSelection = RCC_SMPSCLKSOURCE_HSI;
	PeriphClkInitStruct.SmpsDivSelection = RCC_SMPSCLKDIV_RANGE1;

	if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK) {
		Error_Handler();
	}
	/* USER CODE BEGIN Smps */

	/* USER CODE END Smps */
}

/**
 * @brief I2C1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_I2C1_Init(void) {

	/* USER CODE BEGIN I2C1_Init 0 */

	/* USER CODE END I2C1_Init 0 */

	/* USER CODE BEGIN I2C1_Init 1 */

	/* USER CODE END I2C1_Init 1 */
	hi2c1.Instance = I2C1;
	hi2c1.Init.Timing = 0x00707CBB;
	hi2c1.Init.OwnAddress1 = 0;
	hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
	hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
	hi2c1.Init.OwnAddress2 = 0;
	hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
	hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
	hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
	if (HAL_I2C_Init(&hi2c1) != HAL_OK) {
		Error_Handler();
	}

	/** Configure Analogue filter
	 */
	if (HAL_I2CEx_ConfigAnalogFilter(&hi2c1,
I2C_ANALOGFILTER_ENABLE)
			!= HAL_OK) {
		Error_Handler();
	}

	/** Configure Digital filter
	 */
	if (HAL_I2CEx_ConfigDigitalFilter(&hi2c1, 0) != HAL_OK) {
		Error_Handler();
	}
	/* USER CODE BEGIN I2C1_Init 2 */

	/* USER CODE END I2C1_Init 2 */

}

/**
 * @brief IPCC Initialization Function
```

```
 * @param None
 * @retval None
 */
static void MX_IPCC_Init(void) {

	/* USER CODE BEGIN IPCC_Init 0 */

	/* USER CODE END IPCC_Init 0 */

	/* USER CODE BEGIN IPCC_Init 1 */

	/* USER CODE END IPCC_Init 1 */
	hipcc.Instance = IPCC;
	if (HAL_IPCC_Init(&hipcc) != HAL_OK) {
		Error_Handler();
	}
	/* USER CODE BEGIN IPCC_Init 2 */

	/* USER CODE END IPCC_Init 2 */

}

/**
 * @brief RF Initialization Function
 * @param None
 * @retval None
 */
static void MX_RF_Init(void) {

	/* USER CODE BEGIN RF_Init 0 */

	/* USER CODE END RF_Init 0 */

	/* USER CODE BEGIN RF_Init 1 */

	/* USER CODE END RF_Init 1 */
	/* USER CODE BEGIN RF_Init 2 */

	/* USER CODE END RF_Init 2 */

}

/**
 * @brief RTC Initialization Function
 * @param None
 * @retval None
 */
static void MX_RTC_Init(void) {

	/* USER CODE BEGIN RTC_Init 0 */

	/* USER CODE END RTC_Init 0 */

	/* USER CODE BEGIN RTC_Init 1 */

	/* USER CODE END RTC_Init 1 */

	/** Initialize RTC Only
	 */
	hrtc.Instance = RTC;
	hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
```

```
	hrtc.Init.AsynchPrediv = CFG_RTC_ASYNCH_PRESCALER;
	hrtc.Init.SynchPrediv = CFG_RTC_SYNCH_PRESCALER;
	hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
	hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
	hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
	hrtc.Init.OutPutRemap = RTC_OUTPUT_REMAP_NONE;
	if (HAL_RTC_Init(&hrtc) != HAL_OK) {
		Error_Handler();
	}

	/** Enable the WakeUp
	 */
	if (HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0,
RTC_WAKEUPCLOCK_RTCCLK_DIV16)
			!= HAL_OK) {
		Error_Handler();
	}
	/* USER CODE BEGIN RTC_Init 2 */

	/* USER CODE END RTC_Init 2 */

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void) {
	GPIO_InitTypeDef GPIO_InitStruct = { 0 };
	/* USER CODE BEGIN MX_GPIO_Init_1 */
	/* USER CODE END MX_GPIO_Init_1 */

	/* GPIO Ports Clock Enable */
	__HAL_RCC_GPIOA_CLK_ENABLE();
	__HAL_RCC_GPIOB_CLK_ENABLE();

	/*Configure GPIO pin Output Level */
	HAL_GPIO_WritePin(GPIOA, GPIO_PIN_2, GPIO_PIN_SET);

	/*Configure GPIO pin : PA2 */
	GPIO_InitStruct.Pin = GPIO_PIN_2;
	GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
	GPIO_InitStruct.Pull = GPIO_NOPULL;
	GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
	HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

	/* USER CODE BEGIN MX_GPIO_Init_2 */
	/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
int8_t bme680I2cRead(uint8_t dev_id, uint8_t reg_addr, uint8_t
*reg_data,
		uint16_t len) {
	int8_t result;

	if (HAL_I2C_Master_Transmit(&hi2c1, (dev_id << 1), &reg_addr, 1,
	HAL_MAX_DELAY) != HAL_OK) {
		result = -1;
```

```
      } else if (HAL_I2C_Master_Receive(&hi2c1, (dev_id << 1) | 0x01,
reg_data,
                  len, HAL_MAX_DELAY) != HAL_OK) {
            result = -1;
      } else {
            result = 0;
      }

      return result;
}

int8_t bme680I2cWrite(uint8_t dev_id, uint8_t reg_addr, uint8_t
*reg_data,
            uint16_t len) {
      int8_t result;
      int8_t *buf;

      buf = malloc(len + 1);
      buf[0] = reg_addr;
      memcpy(buf + 1, reg_data, len);

      if (HAL_I2C_Master_Transmit(&hi2c1, (dev_id << 1), (uint8_t*)
buf, len + 1,
      HAL_MAX_DELAY) != HAL_OK) {
            result = -1;
      } else {
            result = 0;
      }

      free(buf);
      return result;
}

uint32_t config_load(uint8_t *config_buffer, uint32_t n_buffer) {
      // ...
      // Load a library config from non-volatile memory, if available.
      //
      // Return zero if loading was unsuccessful or no config was
available,
      // otherwise return length of loaded config string.
      // ...
      return 0;
}

uint32_t state_load(uint8_t *state_buffer, uint32_t n_buffer) {
      // ...
      // Load a previous library state from non-volatile memory, if
available.
      //
      // Return zero if loading was unsuccessful or no state was
available,
      // otherwise return length of loaded state string.
      // ...
      return 0;
}

void sleep(uint32_t t_ms) {
      HAL_Delay(t_ms);
}
/* USER CODE END 4 */
```

```
/**
 * @brief  This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void) {
      /* USER CODE BEGIN Error_Handler_Debug */
      /* User can add his own implementation to report the HAL error
return state */
      __disable_irq();
      while (1) {
      }
      /* USER CODE END Error_Handler_Debug */
}
```

```
#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line
number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and
line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n",
file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

custom_app.c

```
/* USER CODE BEGIN Header */
/**
**********************************************************************
********
 * @file    App/custom_app.c
 * @author  MCD Application Team
 * @brief   Custom Example Application (Server)
**********************************************************************
********
 * @attention
 *
 * Copyright (c) 2024 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the
LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
**********************************************************************
********
```

```
 */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------
---------*/
#include "main.h"
#include "app_common.h"
#include "dbg_trace.h"
#include "ble.h"
#include "custom_app.h"
#include "custom_stm.h"
#include "stm32_seq.h"
#include "LTC2942.h"

/* Private includes ----------------------------------------------------
---------*/
/* USER CODE BEGIN Includes */
#include "bsec_integration.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------
---------*/
typedef struct
{
  /* AirService */
  /* AirService2 */
  /* USER CODE BEGIN CUSTOM_APP_Context_t */

  /* USER CODE END CUSTOM_APP_Context_t */

  uint16_t              ConnectionHandle;
} Custom_App_Context_t;

/* USER CODE BEGIN PTD */
typedef struct {
      int64_t timestamp;
      float iaq;
      uint8_t iaq_accuracy;
      float temperature;
      float humidity;
      float pressure;
      float raw_temperature;
      float raw_humidity;
      float gas;
      bsec_library_return_t bsec_status;
      float static_iaq;
      float co2_equivalent;
      float breath_voc_equivalent;
} output_t;
extern I2C_HandleTypeDef hi2c1;
/* USER CODE END PTD */

/* Private defines -----------------------------------------------------
----------*/
/* USER CODE BEGIN PD */
#define DELAY_PERIOD_MS (1*1000)
/* USER CODE END PD */

/* Private macros ------------------------------------------------------
----------*/
/* USER CODE BEGIN PM */
```

```
/* USER CODE END PM */

/* Private variables ---------------------------------------------------
---------*/
/**
 * START of Section BLE_APP_CONTEXT
 */

static Custom_App_Context_t Custom_App_Context;

/**
 * END of Section BLE_APP_CONTEXT
 */

uint8_t UpdateCharData[247];
uint8_t NotifyCharData[247];

/* USER CODE BEGIN PV */
output_t output;
/* Timestamp variables */
int64_t time_stamp = 0;
int64_t time_stamp_interval_ms = 0;

/* Allocate enough memory for up to BSEC_MAX_PHYSICAL_SENSOR physical
inputs*/
bsec_input_t bsec_inputs[BSEC_MAX_PHYSICAL_SENSOR];

/* Number of inputs to BSEC */
uint8_t num_bsec_inputs = 0;

/* BSEC sensor settings struct */
bsec_bme_settings_t sensor_settings;

/* Save state variables */
uint8_t bsec_state[BSEC_MAX_STATE_BLOB_SIZE];
uint8_t work_buffer[BSEC_MAX_WORKBUFFER_SIZE];
uint32_t bsec_state_len = 0;
uint32_t n_samples = 0;

bsec_library_return_t bsec_status = BSEC_OK;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------
---------*/
/* AirService */
/* AirService2 */

/* USER CODE BEGIN PFP */
void sleep2(uint32_t t_ms);
int64_t get_timestamp_us();
void output_ready(int64_t timestamp, float iaq, uint8_t iaq_accuracy,
            float temperature, float humidity, float pressure,
            float raw_temperature, float raw_humidity, float gas,
            bsec_library_return_t bsec_status, float static_iaq,
            float co2_equivalent, float breath_voc_equivalent);
void state_save(const uint8_t *state_buffer, uint32_t length);

void myTask(void) {
      // Handle battery
      uint8_t all_registers_buffer[1000];
```

```
	uint8_t return_buffer[2];
	read_all_registers(all_registers_buffer, &hi2c1);
	uint8_t status_reg_value =
get_status_reg_value(all_registers_buffer);
	uint8_t control_reg_value =
get_control_reg_value(all_registers_buffer);
	get_charge_reg_value(all_registers_buffer, return_buffer);
	float mAh = get_mAh(return_buffer);
	float battery_lev = get_battery_percentage(mAh);

	if (time_stamp_interval_ms > 0) {

	}else{

	HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_2);

	/* get the timestamp in nanoseconds before calling
bsec_sensor_control() */
	time_stamp = get_timestamp_us() * 1000;

	/* Retrieve sensor settings to be used in this time instant by
calling bsec_sensor_control */
	bsec_sensor_control(time_stamp, &sensor_settings);

	/* Trigger a measurement if necessary */
	bme680_bsec_trigger_measurement(&sensor_settings, sleep2);

	/* Read data from last measurement */
	num_bsec_inputs = 0;
	bme680_bsec_read_data(time_stamp, bsec_inputs, &num_bsec_inputs,
			sensor_settings.process_data);

	/* Time to invoke BSEC to perform the actual processing */
	bme680_bsec_process_data(bsec_inputs, num_bsec_inputs,
output_ready);

	/* Increment sample counter */
	n_samples++;

	/* Retrieve and store state if the passed save_intvl */
	if (n_samples >= 10000) {
		bsec_status = bsec_get_state(0, bsec_state,
sizeof(bsec_state),
				work_buffer, sizeof(work_buffer),
&bsec_state_len);
		if (bsec_status == BSEC_OK) {
			state_save(bsec_state, bsec_state_len);
		}
		n_samples = 0;
	}

	}

	char temp[4];

	sprintf(temp, "%.2f", output.temperature);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRTEMP, &temp);

	char pres[10];
```

```
	sprintf(pres, "%.0f", output.pressure);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRPRES, &pres);

	char hum[4];

	sprintf(hum, "%.2f", output.humidity);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRHUM, &hum);

	char voc[10];

	sprintf(voc, "%.1f", output.breath_voc_equivalent);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRVOCS, &voc);

	char co2[10];

	sprintf(co2, "%.0f", output.co2_equivalent);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRCO2, &co2);

	char iaq_acc[1];

	sprintf(iaq_acc, "%d", output.iaq_accuracy);

	Custom_STM_App_Update_Char(CUSTOM_STM_IAQACC, &iaq_acc);

	char iaq[2];

	sprintf(iaq, "%.0f", output.iaq);

	Custom_STM_App_Update_Char(CUSTOM_STM_AIRIAQ, &iaq);

	/* Compute how long we can sleep until we need to call
bsec_sensor_control() next */
	/* Time_stamp is converted from microseconds to nanoseconds
first and then the difference to milliseconds */
	time_stamp_interval_ms = (sensor_settings.next_call
			- get_timestamp_us() * 1000) / 1000000;

	char batt[4];

	sprintf(batt, "%.2f", battery_lev);

	Custom_STM_App_Update_Char(CUSTOM_STM_BATT, &batt);

	UTIL_SEQ_SetTask(1 << CFG_TASK_MY_TASK, CFG_SCH_PRIO_0);
}
/* USER CODE END PFP */

/* Functions Definition ----------------------------------------------
---------*/
void Custom_STM_App_Notification(Custom_STM_App_Notification_evt_t
*pNotification)
{
  /* USER CODE BEGIN CUSTOM_STM_App_Notification_1 */

  /* USER CODE END CUSTOM_STM_App_Notification_1 */
  switch (pNotification->Custom_Evt_Opcode)
  {
```

```
    /* USER CODE BEGIN CUSTOM_STM_App_Notification_Custom_Evt_Opcode
*/

    /* USER CODE END CUSTOM_STM_App_Notification_Custom_Evt_Opcode */

    /* AirService */
    case CUSTOM_STM_AIRTEMP_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRTEMP_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRTEMP_READ_EVT */
      break;

    case CUSTOM_STM_AIRPRES_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRPRES_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRPRES_READ_EVT */
      break;

    case CUSTOM_STM_AIRHUM_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRHUM_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRHUM_READ_EVT */
      break;

    case CUSTOM_STM_AIRVOCS_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRVOCS_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRVOCS_READ_EVT */
      break;

    case CUSTOM_STM_AIRCO2_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRCO2_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRCO2_READ_EVT */
      break;

    /* AirService2 */
    case CUSTOM_STM_IAQACC_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_IAQACC_READ_EVT */

      /* USER CODE END CUSTOM_STM_IAQACC_READ_EVT */
      break;

    case CUSTOM_STM_AIRIAQ_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_AIRIAQ_READ_EVT */

      /* USER CODE END CUSTOM_STM_AIRIAQ_READ_EVT */
      break;

    case CUSTOM_STM_BATT_READ_EVT:
      /* USER CODE BEGIN CUSTOM_STM_BATT_READ_EVT */

      /* USER CODE END CUSTOM_STM_BATT_READ_EVT */
      break;

    case CUSTOM_STM_NOTIFICATION_COMPLETE_EVT:
      /* USER CODE BEGIN CUSTOM_STM_NOTIFICATION_COMPLETE_EVT */

      /* USER CODE END CUSTOM_STM_NOTIFICATION_COMPLETE_EVT */
      break;
```

```
    default:
      /* USER CODE BEGIN CUSTOM_STM_App_Notification_default */

      /* USER CODE END CUSTOM_STM_App_Notification_default */
      break;
  }
  /* USER CODE BEGIN CUSTOM_STM_App_Notification_2 */

  /* USER CODE END CUSTOM_STM_App_Notification_2 */
  return;
}

void Custom_APP_Notification(Custom_App_ConnHandle_Not_evt_t
*pNotification)
{
  /* USER CODE BEGIN CUSTOM_APP_Notification_1 */

  /* USER CODE END CUSTOM_APP_Notification_1 */

  switch (pNotification->Custom_Evt_Opcode)
  {
    /* USER CODE BEGIN CUSTOM_APP_Notification_Custom_Evt_Opcode */

    /* USER CODE END P2PS_CUSTOM_Notification_Custom_Evt_Opcode */
    case CUSTOM_CONN_HANDLE_EVT :
      /* USER CODE BEGIN CUSTOM_CONN_HANDLE_EVT */

      /* USER CODE END CUSTOM_CONN_HANDLE_EVT */
      break;

    case CUSTOM_DISCON_HANDLE_EVT :
      /* USER CODE BEGIN CUSTOM_DISCON_HANDLE_EVT */

      /* USER CODE END CUSTOM_DISCON_HANDLE_EVT */
      break;

    default:
      /* USER CODE BEGIN CUSTOM_APP_Notification_default */

      /* USER CODE END CUSTOM_APP_Notification_default */
      break;
  }

  /* USER CODE BEGIN CUSTOM_APP_Notification_2 */

  /* USER CODE END CUSTOM_APP_Notification_2 */

  return;
}

void Custom_APP_Init(void)
{
  /* USER CODE BEGIN CUSTOM_APP_Init */

  /* USER CODE END CUSTOM_APP_Init */
  return;
}

/* USER CODE BEGIN FD */
/*void sleep(uint32_t t_ms) {
 HAL_Delay(t_ms);
```

```
 }*/

int64_t get_timestamp_us() {
      int64_t system_current_time = 0;
      uint32_t tick;
      tick = HAL_GetTick();
      system_current_time = 1000 * (int64_t) tick;
      return system_current_time;
}

void output_ready(int64_t timestamp, float iaq, uint8_t iaq_accuracy,
            float temperature, float humidity, float pressure,
            float raw_temperature, float raw_humidity, float gas,
            bsec_library_return_t bsec_status, float static_iaq,
            float co2_equivalent, float breath_voc_equivalent) {

      output.timestamp = timestamp;
      output.iaq = iaq;
      output.iaq_accuracy = iaq_accuracy;
      output.temperature = temperature;
      output.humidity = humidity;
      output.pressure = pressure;
      output.raw_temperature = raw_temperature;
      output.raw_humidity = raw_humidity;
      output.gas = gas;
      output.bsec_status = bsec_status;
      output.static_iaq = static_iaq;
      output.co2_equivalent = co2_equivalent;
      output.breath_voc_equivalent = breath_voc_equivalent;

}

void state_save(const uint8_t *state_buffer, uint32_t length) {
      // ...
      // Save the string some form of non-volatile memory, if
possible.
      // ...
}

void sleep2(uint32_t t_ms) {
      HAL_Delay(t_ms);
}
/* USER CODE END FD */

/*************************************************************
 *
 * LOCAL FUNCTIONS
 *
 *************************************************************/

/* AirService */
/* AirService2 */

/* USER CODE BEGIN FD_LOCAL_FUNCTIONS*/
/* USER CODE END FD_LOCAL_FUNCTIONS*/
```

LTC2942.c

```
/*
 * LTC2942.c
 *
 *  Created on: 2nd Apr 2024
 *      Author: Dídac Roda Pitarg
 *
 *      @description    library written for LTC2942-1 LiIon charge
estimator
 *
 */

#include "main.h"
#include "stm32wbxx_hal.h"
#include "LTC2942.h"

static int prescalar = 128;
static int maxAmh = 5570;
const float q_LSB = 0.085; // mAh

/*
 * @brief               function gets all registers because of strange
incrementation of address in case of getting single reg
 * @param               buffer which will contain 2 bytes of the data
 * @return              function returns control register value via
pointer
 */
void read_all_registers(uint8_t *buf, I2C_HandleTypeDef *hi2c) {
	while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
		;
	HAL_I2C_Master_Transmit(hi2c, LTC2942_ADDR, 0x00, 1, 1000);
	while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
		;
	HAL_I2C_Master_Receive(hi2c, LTC2942_ADDR, buf, 8, 1000);
	while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
		;

}

/*
 * @brief               function gets status reg value from all
registers value
 * @param               all registers value buffer
@read_all_registers()
 * @return              function returns 1 byte status reg value
 */
uint8_t get_status_reg_value(uint8_t *all_reg_buf) {
	return all_reg_buf[0];
}

/*
 * @brief               function gets control reg value from all
registers value
 * @param               all registers value buffer
@read_all_registers()
 * @return              function returns 1 byte control reg value
 */
uint8_t get_control_reg_value(uint8_t *all_reg_buf) {
	return all_reg_buf[1];
}
```

```
/*
 * @brief                  function gets charge reg value from all
registers value
 * @param                  all registers value buffer
@read_all_registers()
 * @return                 function returns 2 byte charge reg value via
pointer
 */
void get_charge_reg_value(uint8_t *all_reg_buf, uint8_t *return_buf) {
      return_buf[0] = all_reg_buf[2];
      return_buf[1] = all_reg_buf[3];
}

/*
 * @brief                  function gets high threshold reg value from all
registers value
 * @param                  all registers value buffer
@read_all_registers()
 * @return                 function returns 2 byte high threshold reg
value via pointer
 */
void get_high_thresh_reg_value(uint8_t *all_reg_buf, uint8_t
*return_buf) {
      return_buf[0] = all_reg_buf[4];
      return_buf[1] = all_reg_buf[5];
}

/*
 * @brief                  function gets low threshold reg value from all
registers value
 * @param                  all registers value buffer
@read_all_registers()
 * @return                 function returns 2 byte low threshold reg value
via pointer
 */
void get_low_thresh_reg_value(uint8_t *all_reg_buf, uint8_t
*return_buf) {
      return_buf[0] = all_reg_buf[6];
      return_buf[1] = all_reg_buf[7];
}

/*
 * @brief                  function gets amount of charge from accumulated
charge register (register C and D)
 * @param                  accumulated charge register value buffer
@get_charge_reg_value()
 * @return                 function returns mAh based on the prescalar and
accumulated charge
 */
float get_mAh(uint8_t *acc_charge_reg_buf) {
      uint16_t data = (acc_charge_reg_buf[1] | (acc_charge_reg_buf[0]
<< 8)); // 0 is MSB and 1 is LSB
      float mAh = (float) (data * q_LSB * prescalar) / (128);

      return mAh;
}

/*
 * @brief                  function gets amount of charge in percentage
 * @param                  amount of charge value buffer @get_mAh()
```

```
 * @return                function returns mAh in % based on the maxAmh
value from the battery
 */
float get_battery_percentage(float mAh) {
      return (mAh * 100.0f / maxAmh);
}

/*
 * @brief                 function sets maximum value for battery mAh and
prescalar
 * @param                 Handle Type Def of I2C and mAh value of the
battery
 * @return                no value returned
 */
void set_battery_full_mah(I2C_HandleTypeDef *hi2c, uint16_t mAh) {
      uint8_t i = 0;
      float qLSB = 0, temp = mAh;

      if (mAh > maxAmh) {
            return;
      }

      qLSB = temp / 65536;

      uint8_t prescalarTable[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };

      for (i = 0; i < 8; i++) {
            temp = (q_LSB * prescalarTable[i]) / (128);
            if (qLSB <= temp) {
                  break;
            }
      }

      if (i >= 8) {
            return;
      }

      maxAmh = mAh;

      prescalar = prescalarTable[i];

      qLSB = (q_LSB * prescalar) / (128);

      temp = mAh;
      temp = temp / qLSB;
      mAh = (uint16_t) temp;
}

/*
 * @brief                 function sets maximum value 0xFFFF of charge
register (0x02 ... 0x03)
 * @param                 Handle Type Def of I2C
 * @return                no value returned
 */
void set_charge_value_max(I2C_HandleTypeDef *hi2c) {
      uint8_t buf[3];
      buf[0] = ACC_CHARGE_MSB;
      buf[1] = 0xFF;
      buf[2] = 0xFF;

      while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
```

```
        ;
    HAL_I2C_Master_Transmit_IT(hi2c, LTC2942_ADDR, buf, 3);
    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
}

/*
 * @brief               function sets minimum value 0x0000 of charge
register (0x02 ... 0x03)
 * @param               Handle Type Def of I2C
 * @return              no value returned
 */

void set_charge_value_min(I2C_HandleTypeDef *hi2c) {
    uint8_t buf[3];
    buf[0] = ACC_CHARGE_MSB;
    buf[1] = 0x00;
    buf[2] = 0x00;

    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
    HAL_I2C_Master_Transmit_IT(hi2c, LTC2942_ADDR, buf, 3);
    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
}

/*
 * @brief               function sets desired value of charge register
(0x02 ... 0x03)
 * @param               Handle Type Def of I2C
 * @param               buffer of charge value register to be written
 * @return              no value returned
 */
void set_charge_value(I2C_HandleTypeDef *hi2c, uint8_t
*buf_charge_value) {
    uint8_t buf[3];
    buf[0] = ACC_CHARGE_MSB;
    buf[1] = buf_charge_value[0];
    buf[2] = buf_charge_value[1];

    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
    HAL_I2C_Master_Transmit_IT(hi2c, LTC2942_ADDR, buf, 3);
    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
}

/*
 * @brief               function sets desired value of control register
 * @param               Handle Type Def of I2C
 * @param               buffer of control value register to be written
 * @return              no value returned
 */
void set_control_reg_value(I2C_HandleTypeDef *hi2c, uint8_t
buf_charge_value) {
    uint8_t buf[2];
    buf[0] = CONTROL_REG;
    buf[1] = buf_charge_value;

    while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
        ;
```

```
	HAL_I2C_Master_Transmit(hi2c, LTC2942_ADDR, buf, 2, 1000);
	while (HAL_I2C_GetState(hi2c) != HAL_I2C_STATE_READY)
		;
}
```

Appendix III – BLE App Inventor Android app

```
initialize global valors6 to create empty list
initialize global valor6 to " "
initialize global valor5 to " "
initialize global valor4 to " "
initialize global valor3 to " "
initialize global valor2 to " "
initialize global valors5 to create empty list
initialize global valors4 to create empty list
initialize global valors3 to create empty list
initialize global valors2 to create empty list
initialize global valors to create empty list
initialize global Address to " "
initialize global contador to 0

when ScanButton .Click
do  set MessageLabel . Text to " Scanning "
    call BluetoothLE1 .StartScanning

initialize global readready to false

when ButtonConnect .Click
do  call BluetoothLE1 .StopScanning
    set MessageLabel . Text to " Connecting... "
    call BluetoothLE1 .ConnectWithAddress
                                address get global Address
    set ButtonDisconnect . Enabled to false

when ActivityStarter1 .AfterActivity
  result
do  set MessageLabel . Text to " Bluetooth is enabled. Press Scan button. "

when ButtonStop .Click
do  set global readready to false
    set global contador to 0
    set MessageLabel . Text to " Stop measure "
```

```
when Screen1 .Initialize
do  if BluetoothClient1 . Enabled
    then set MessageLabel . Text to " Bluetooth is enabled.Press Scan button. "
    else set ActivityStarter1 . Action to " android.bluetooth.adapter.action.REQUEST_ENABLE "
         call ActivityStarter1 .StartActivity
         call TinyDB1 .StoreValue
                          tag " SaveToIOT "
                 valueToStore false
         call TinyDB1 .StoreValue
                          tag " SaveToLogFile "
                 valueToStore false
         call TinyDB1 .StoreValue
                          tag " SaveToURV "
                 valueToStore false
         call TinyDB1 .StoreValue
                          tag " SamplingPeriod "
                 valueToStore 2

when ButtonConnect .Click
do  call BluetoothLE1 .StopScanning
    set MessageLabel . Text to " Connecting... "
    call BluetoothLE1 .ConnectWithAddress
                         address get global Address
    set ButtonDisconnect . Enabled to false

when ActivityStarter1 .AfterActivity
 result
do  set MessageLabel . Text to " Bluetooth is enabled. Press Scan button. "

when ButtonStop .Click
do  set global readready to false
    set global contador to 0
    set MessageLabel . Text to " Stop measure "
```

```
when BluetoothLE1 .DeviceFound
do  set MessageLabel . Text to " Devices found. Select a device from list "
    set ListPicker1 . ElementsFromString to BluetoothLE1 . DeviceList
    set ListPicker1 . Enabled to true
```

```
when ButtonDisconnect .Click
do  call BluetoothLE1 .DisconnectWithAddress
                                address  get global Address
    set MessageLabel . Text to " Disconnecting... "
```

```
when ButtonPlot .Click
do  call ChartMaker1 .DrawLineGraph
                       chartTitle  " Environmental Measurements "
                       hAxisTitle  " Mesurement "
                       vAxisTitle  " bpm "
                           labels  make a list " Temp (°C) "
                           values  get global valors
                        webViewer  WebViewer1
```

```
when ActivityStarter1 .ActivityCanceled
do  set MessageLabel . Text to " Action was canceled "
```

```
when BluetoothLE1 .Connected
do  set MessageLabel . Text to join " Connected to: "
                                    get global Address
    set ButtonDisconnect . Enabled to true
    set ButtonRead . Enabled to true
    set ButtonStop . Enabled to false
                                  Returns the boolean false.
```

```
when ListPicker1 .AfterPicking
do  set global Address to select list item list split at spaces ListPicker1 . Selection
                                          index 1
    set global DeviceName to select list item list split at spaces ListPicker1 . Selection
                                             index 2
    set ButtonConnect . Enabled to true
    set MessageLabel . Text to get global DeviceName
```

```
when BluetoothLE1 .Disconnected
do  set MessageLabel . Text to "Disconnected"
    set ButtonDisconnect . Enabled to false
    set ButtonRead . Enabled to false
    set ButtonStop . Enabled to false
    set global readready to false

when ConfigButton .Click
do  open another screen screenName Screen2

initialize global DeviceName to " "

when ButtonRead .Click
do  set global readready to true
    set global valors to create empty list
    set global valors2 to create empty list
    set global valors3 to create empty list
    set global valors4 to create empty list
    set global valors5 to create empty list
    set ButtonStop . Enabled to true
    set MessageLabel . Text to "Starting measure"
    set Clock1 . TimerInterval to 1000 × call TinyDB1 .GetValue
                                              tag "SamplingPeriod"
                                              valueIfTagNotThere 2

when BluetoothLE1 .StringsReceived
serviceUuid  characteristicUuid  stringValues
do  if get global readready
    then set global contador to get global contador + 1
         if get serviceUuid = "00000000-cc7a-482a-984a-7f2ed5b3e58f"
         then if get characteristicUuid = "00000000-8e22-4541-9d4c-21edae82ed19"
              then set global valor to select list item list get stringValues
                                                        index 1
                   set MessageLabel . Text to get global valor
                   if call TinyDB1 .GetValue
                           tag "SaveToLogFile"
                           valueIfTagNotThere true
                   then call File1 .AppendToFile
                             text join get global valor
                                       "\n"
                             fileName "/BLEtemp.txt"
                   add items to list list get global valors
                                     item make a list get global contador
                                                      get global valor
```

```
if get characteristicUuid = "00000001-8e22-4541-9d4c-21edae82ed19"
then set global valor2 to select list item list get stringValues
                                           index 1
     set MessageLabel . Text to get global valor2
     if call TinyDB1 .GetValue
                     tag "SaveToLogFile"
         valueIfTagNotThere true
     then call File1 .AppendToFile
                     text join get global valor2
                               "\n"
                     fileName "/BLEpres.txt"
     add items to list list get global valors2
                       item make a list get global contador
                                        get global valor2

if get characteristicUuid = "00000010-8e22-4541-9d4c-21edae82ed19"
then set global valor3 to select list item list get stringValues
                                           index 1
     set MessageLabel . Text to get global valor3
     if call TinyDB1 .GetValue
                     tag "SaveToLogFile"
         valueIfTagNotThere true
     then call File1 .AppendToFile
                     text join get global valor3
                               "\n"
                     fileName "/BLEhum.txt"
     add items to list list get global valors3
                       item make a list get global contador
                                        get global valor3

if get characteristicUuid = "00000011-8e22-4541-9d4c-21edae82ed19"
then set global valor4 to select list item list get stringValues
                                           index 1
     set MessageLabel . Text to get global valor4
     if call TinyDB1 .GetValue
                     tag "SaveToLogFile"
         valueIfTagNotThere true
     then call File1 .AppendToFile
                     text join get global valor5
                               "\n"
                     fileName "/BLEbVOC.txt"
     add items to list list get global valors4
                       item make a list get global contador
                                        get global valor4
```

```
if  get characteristicUuid = "00000100-8e22-4541-9d4c-21edae82ed19"
then  set global valor5 to  select list item  list  get stringValues
                                              index  1
      set MessageLabel . Text to  get global valor5
      if  call TinyDB1 .GetValue
                         tag  "SaveToLogFile"
          valueIfTagNotThere  true
      then  call File1 .AppendToFile
                         text  join  get global valor5
                                     "\n"
                     fileName  "/BLECO2.txt"
      add items to list  list  get global valors5
                         item  make a list  get global contador
                                            get global valor5

if  get serviceUuid = "00000001-cc7a-482a-984a-7f2ed5b3e58f"
then  if  get characteristicUuid = "00000001-8e22-4541-9d4c-21edae82ed19"
      then  set global valor6 to  select list item  list  get stringValues
                                                    index  1
            set MessageLabel . Text to  get global valor6
            if  call TinyDB1 .GetValue
                               tag  "SaveToLogFile"
                valueIfTagNotThere  true
            then  call File1 .AppendToFile
                               text  join  get global valor6
                                           "\n"
                           fileName  "/BLEIAQ.txt"
            add items to list  list  get global valors6
                               item  make a list  get global contador
                                                  get global valor6

set Web1 . Url to  join  join  get global ThingsPeakGetString
                               get global valor
                         join  join  "&field2="
                                     get global valor2
                               join  join  "&field3="
                                           get global valor3
                                     join  join  "&field4="
                                                 get global valor4
                                           join  join  "&field5="
                                                       get global valor5
                                                 join  "&field6="
                                                       get global valor6
call Web1 .Get
```

```
initialize global ThingsPeakGetString to " https://api.thingspeak.com/update?api_key=CL51Q6... "

initialize global URVsPeakGetString to " http://130.206.36.41/ioturv/view/setdadesg.php?w... "

when Clock1 .Timer
do  if  get global readready
    then  if  BluetoothLE1 . IsDeviceConnected
          then  call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000000-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000000-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
                call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000000-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000001-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
                call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000000-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000010-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
                call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000000-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000011-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
                call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000000-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000100-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
                call BluetoothLE1 .ReadStrings
                     serviceUuid        " 00000001-CC7A-482A-984A-7F2ED5B3E58F "
                     characteristicUuid " 00000001-8E22-4541-9D4C-21EDAE82ED19 "
                     utf16              false
```