ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
UNIVERSITY OF PIRAEUS

Department of Digital Systems

Thesis

of

Dimou Orfeas
mwe1901

on the

Design and Development of an IoT System for Indoor Air Quality Estimation Using Low-Power Hardware and Bluetooth Technologies

Supervising Committee:	Dr. Antonis Gotsis (External)
	Prof. Athanasios Kanatas
	Prof. George Efthymoglou

March 2021

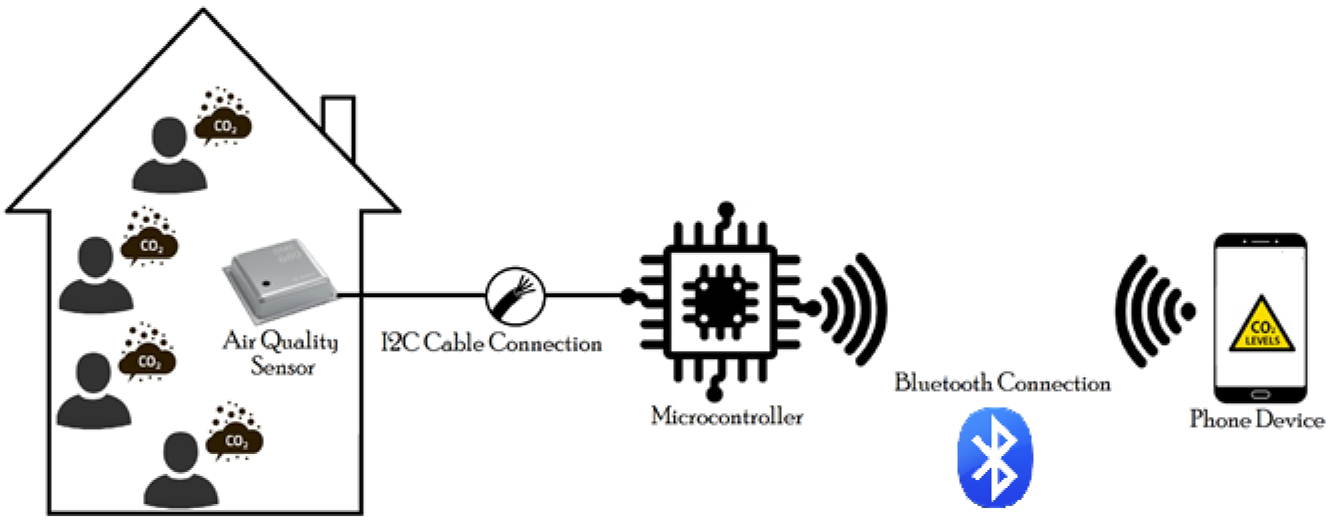
Introduction

The main objective of the Thesis is to design and develop an IoT System that is able to estimate Indoor Air Quality, in other words, CO2 levels. As humans exhale CO2, CO2 levels are higher in Indoor spaces where air cannot escape fast. Taking that into consideratio, approximately all people exhale almost the same amount of CO2, the level of CO2 in an Indoor space is proportional to the number of people inside. That said, this IoT system will be able to measure the CO2 level and convert it into number of people inside. This System could prove very useful in areas with lots of people, as it is able to calculate the average number of them without performing a headcount, thus saving as time. Also, the ability to calculate people fast, can help to prevent overcrowding and maintain health restrictions about max number of persons in a given space.

To create such a system, we will be using an VOC sensor (BME680) for acquiring CO2-equivalent levels, a microcontroller (STM32WB55) to process/export the data acquired from the sensor and a phone to collect transmitted data. Since we want flexibility and an easy installation/usage, the system will be powered with batteries and the data will be transmitted by making use of, the Bluetooth Technology. The microcontroller that we have chosen to use, combines Bluetooth and low power consumption capabilities.

In the following Chapters we will analyze the theory behind IoT, BLE, I2C and show the proper way to program, use, and acquire data to our phone from the STM32WB55.

Finally, through experiments that we have performed, we will provide some indicative results and how we can use them.



Contents

Chapter 1: Background

1.1 The IoT Concept

There is a lot of talk at the moment, about the Internet of Things (IoT) and its impact on everything. But what is the Internet of Things? How does it work? And is it really that important?

In a nutshell, the Internet of Things is the concept of connecting any device to the Internet and to other connected devices, like sensors, software, and other technologies for the purpose of exchanging data. These devices range from ordinary household objects to sophisticated industrial tools. The IoT is a giant network of connected things and people – all of which collect and share data about the way they are used and about the environment around them.

The way that it works is that devices and objects with built in sensors are connected to an Internet of Things platform which integrates data from the different devices and applies analytics to share the most valuable information with applications built to address specific needs.

By the word 'Things' we refer to machines or physical objects, so it becomes important to understand what kind of objects can be connected via Internet. We can categories these objects into categories –

1. Objects with intelligence or Smart Objects.
   Smart Object: "Smart objects are those physical and digital objects which can be identified, have sensing/actuating capabilities, computational power, also storing, and networking capabilities."
2. Objects without intelligence or Non-Smart Objects.
   Non-Smart Objects: Non-smart objects are generally those objects which do not have intelligence and processing capabilities. Sensors and actuators are non-smart devices.

But why Is Internet of Things (IoT) so important?

Over the past few years, IoT has become one of the most important technologies of the 21st century. Now that we can connect everyday objects to the internet via embedded devices, seamless communication is possible between people, processes, and things. By means of low-cost computing, the cloud, big data, analytics, and mobile technologies, physical things can share and collect data with minimal human intervention.

While the idea of IoT has been in existence for a long time, a collection of recent advances in several different technologies has made it practical.

- Access to low-cost, low-power sensor technology. Affordable and reliable sensors are making IoT technology possible for more manufacturers.
- Connectivity. A host of network protocols for the internet has made it easy to connect sensors to the cloud and to other "things" for efficient data transfer.
- Cloud computing platforms. The increase in the availability of cloud platforms enables both businesses and consumers to access the infrastructure they need to scale up without having to manage it all.
- Machine learning and analytics. With advances in machine learning and analytics, along with access to varied and vast amounts of data stored in the cloud, businesses can gather insights faster and more easily. The emergence of these allied technologies continues to push the boundaries of IoT and the data produced by IoT also feeds these technologies.
- Conversational artificial intelligence (AI). Advances in neural networks have brought natural-language processing (NLP) to IoT devices and made them appealing, affordable, and viable for home use.

1.2 The Bluetooth

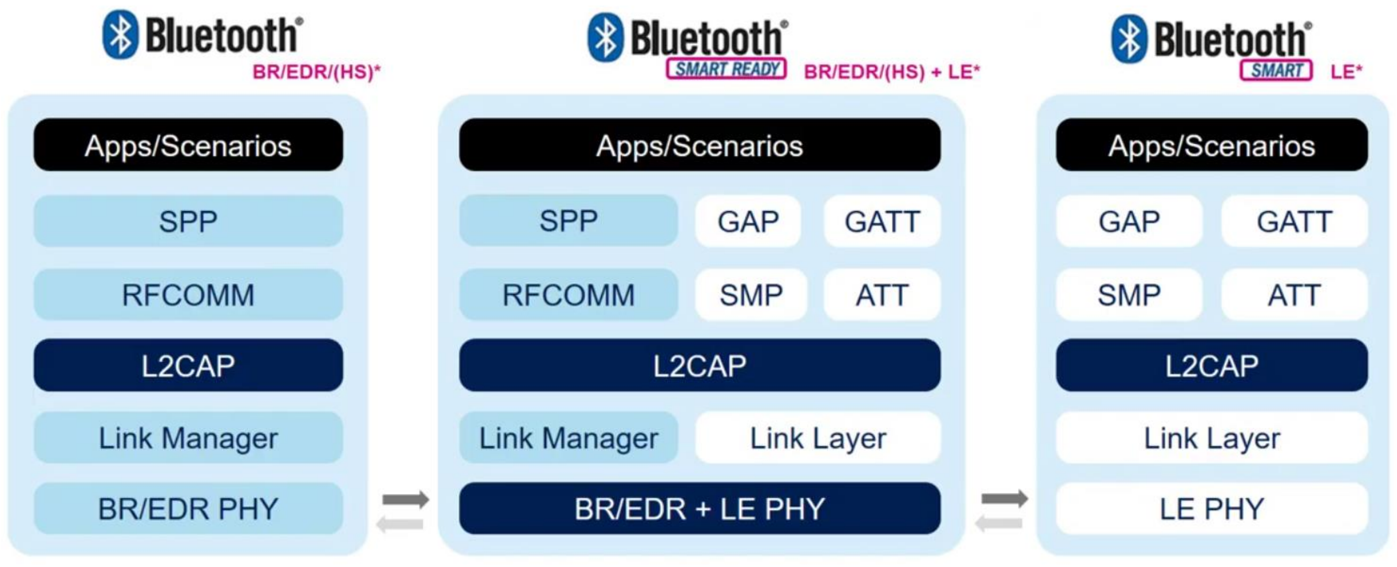
Bluetooth wireless technology (BWT) was developed in 1994 at Ericsson in Sweden. The original purpose of BWT was to eliminate the need for proprietary cable connections between devices such as RS-232 data cables.

BWT-enabled devices operate in the unrestricted 2.4-gigahertz (GHz) Industrial, Science, Medical (ISM) band. The ISM band ranges between 2.400 GHz and 2.483 GHz (ISM Band). Bluetooth sends and receives radio waves in a band of 79 different frequencies (channels) centered on 2.45 GHz, set apart from radio, television, and cellphones, and reserved for use by industrial, scientific, and medical gadgets. BWT-enabled devices use a technique called frequency hopping to minimize eavesdropping and interference from other networks that use the ISM band. With frequency hopping, the data is divided into small pieces called packets. The transmitter and receiver exchange a data packet at one frequency, and then they hop to another frequency to exchange another packet. They repeat this process until all the data is transmitted.

Bluetooth is a radio-wave technology, mainly designed for communicating over short distances less than about 10m or 30ft.

This IoT protocol (Bluetooth Low Energy) brings the protocol on a new level. It opens new opportunities for devices with small battery capacity. However, the range of this protocol is even less than Wi-Fi has. Besides, the data exchange speed is suitable only for small sized data. Minding these facts, we can see that Bluetooth is a perfect option for wearable devices.

At the next Figure we can see the different Protocol stacks of Bluetooth. The Bluetooth is a collection of different protocols grouped together under a single specification.



The first Protocol stack is the HS (High Speed) Bluetooth which uses the protocol SPP (Serial Peripheral Protocol) and it is one of the first Protocols that were used. In the second Protocol stack, the Smart Ready-LE (Low Energy), is the one that most mobiles, laptops, and tablets use. It consists of protocols of both Protocol stacks as it can communicate with both. Also, the devices that use it have enough memory space to include all protocols. Finally, the third Protocol stack (Smart- LE (Low Energy)) is the one that we are going to use in this project.

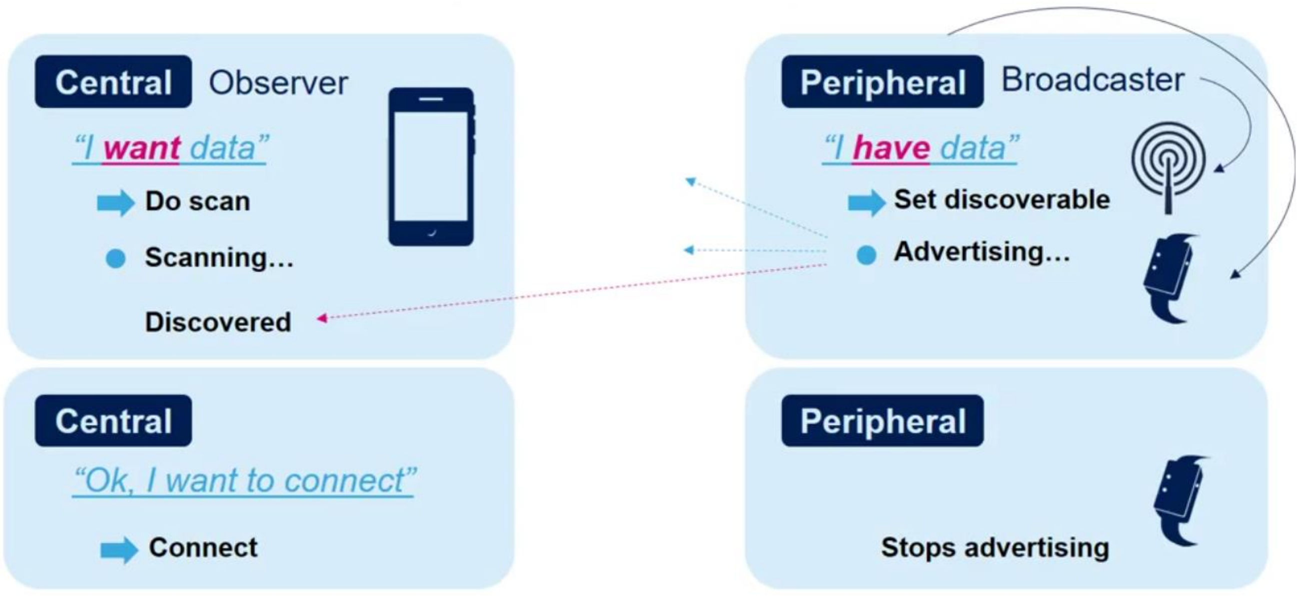
The core specifications of the device that we use is the Bluetooth 5 that is available since 2016. Our device is compatible with version 5 but it doesn’t have all the new functions and features that were added.

Focusing on the protocols of the Bluetooth smart:

GAP: Generic Access Profile

Everything on both ends of the communicating devices start with this protocol.

- The Gap layer controls advertising and connections (makes a device visible to the outside world)
- Also determines how two devices can interact with each other.

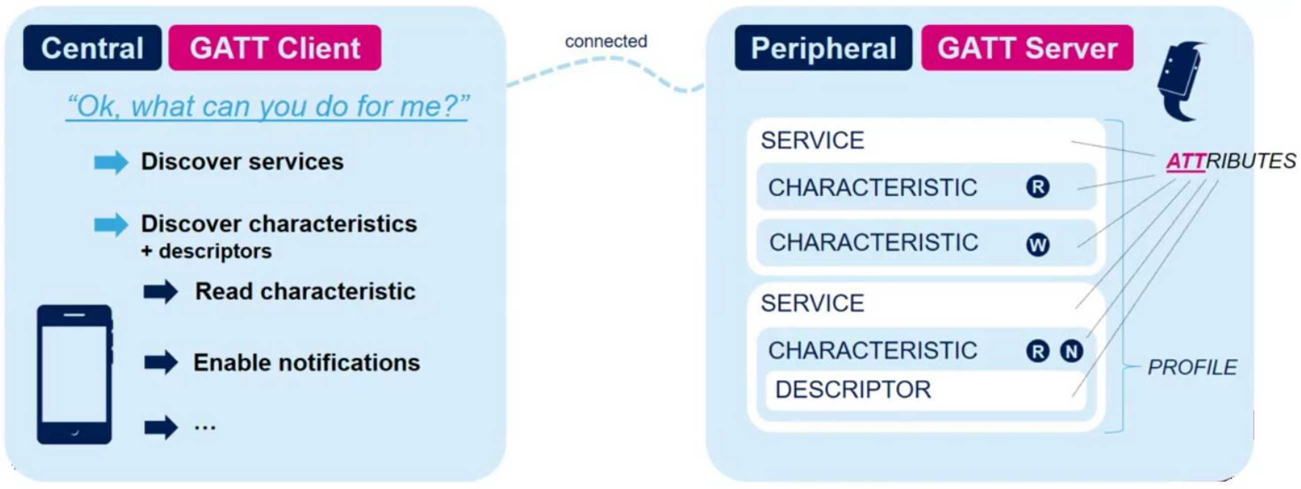


While advertising the packet transmitted includes information of the data that are going to be transmitted once connected.

Once we establish the connection, we can move on to the GATT protocol.

GATT: Generic Attribute Profile

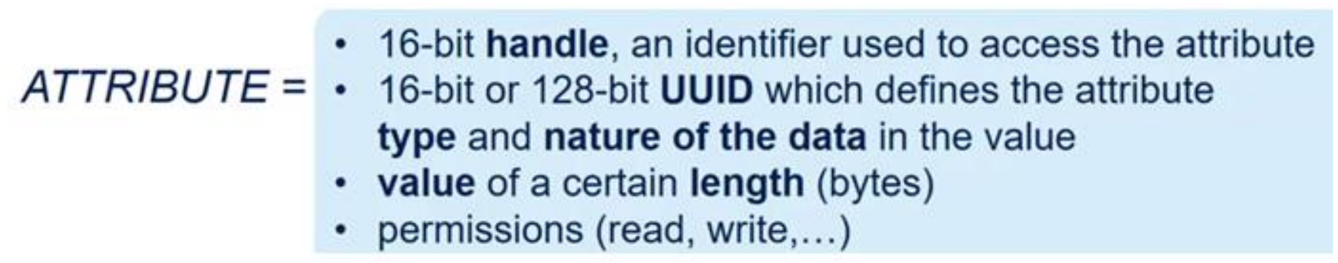
In this step our Central asks the peripheral about what services it offers and their characteristics (like read, write, notification etc).



So, our Central device reads the service and gets a response of what the service or characteristics are. Also, the Central can write to the Peripheral and get a response.

ATT: Attribute Protocol

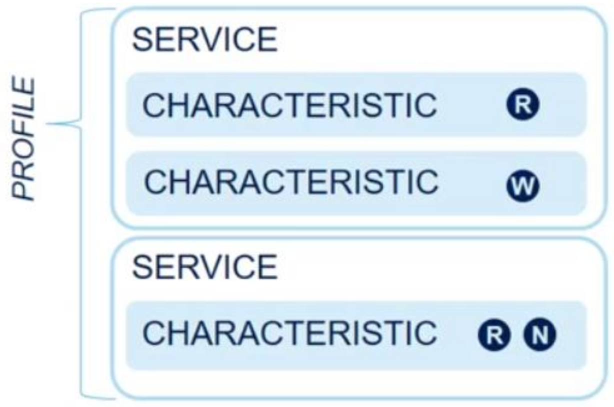
This protocol defines what the communication is going to be between client and server. In our case the server is the peripheral. The Attributes are stored in the server and listed as tables. The Attributes contain lots of information, like ID's, parameters, data length etc.



ATTRIBUTE =
- 16-bit **handle**, an identifier used to access the attribute
- 16-bit or 128-bit **UUID** which defines the attribute **type** and **nature of the data** in the value
- **value** of a certain **length** (bytes)
- permissions (read, write,…)

So, ATT is just an array of bytes stored in a table, data logic and hierarchy given by GATT and app layer.

Summarizing, when the GAP procedure finishes and the devices are ready to communicate, the GATT comes into play and a connection is established and defines data exchange between two BLE devices. It adds a data model and hierarchy on top of the ATT (by means of concepts called services and characteristics). The services are organized in GATT profiles and each profile can contain multiple services.



- A service is a container for logically related data items
- Characteristics are logically related data items within one service and consist of a type, a value, some properties, permissions and optionally descriptors.
- Descriptors either provides additional details or allows configuration of behavior related to the characteristics (e.g., turn on notifications)

1.3 The Microcontrollers

A microcontroller (µC or uC) is a solitary chip microcomputer fabricated from VLSI fabrication. A micro controller is also known as embedded controller. Today various types of microcontrollers are available in market with different word lengths such as 4bit, 8bit, 64bit and 128bit microcontrollers. Microcontroller is a compressed micro computer manufactured to control the functions of embedded systems in office machines, robots, home appliances, motor vehicles, and a number of other gadgets. A microcontroller includes components like – memory, peripherals and most importantly a processor. Microcontrollers are basically employed in devices that need a degree of control to be applied by the user of the device.

Types of Microcontrollers

Microcontrollers are divided into various categories based on memory, architecture, bits and instruction sets. Following is the list of their types

Bit

Based on bit configuration, the microcontroller is further divided into three categories.

- 8-bit microcontroller − This type of microcontroller is used to execute arithmetic and logical operations like addition, subtraction, multiplication division, etc.
- 16-bit microcontroller − This type of microcontroller is used to perform arithmetic and logical operations where higher accuracy and performance is required
- 32-bit microcontroller − This type of microcontroller is generally used in automatically controlled appliances like automatic operational machines, medical appliances, etc.

Memory

Based on the memory configuration, the microcontroller is further divided into two categories.

- External memory microcontroller − This type of microcontroller is designed in such a way that they do not have a program memory on the chip. Hence, it is named as external memory microcontroller.
- Embedded memory microcontroller − This type of microcontroller is designed in such a way that the microcontroller has all programs and data memory, counters and timers, interrupts, I/O ports are embedded on the chip.

Instruction Set

Based on the instruction set configuration, the microcontroller is further divided into two categories.

- CISC − CISC stands for complex instruction set computer. It allows the user to insert a single instruction as an alternative to many simple instructions.
- RISC − RISC stands for Reduced Instruction Set Computers. It reduces the operational time by shortening the clock cycle per instruction.

Microcontroller Basics:

Any electric appliance that stores, measures, displays information or calculates comprise of a microcontroller chip inside it. The basic structure of a microcontroller comprises of:

CPU: Microcontrollers brain is named as CPU. CPU is the device, which is employed to fetch data, decode it and at the end complete the assigned task successfully. With the help of CPU all the components of microcontroller are connected into a single system. Instruction fetched by the programmable memory is decoded by the CPU.

Memory: In a microcontroller memory chip works same as microprocessor. Memory chip stores all programs & data. Microcontrollers are built with certain amount of ROM or RAM (EPROM, EEPROM, etc) or flash memory for the storage of program source codes.

Input/output ports: I/O ports are basically employed to interface or drive different appliances such as- printers, LCD's, LED's, etc.

Serial Ports: These ports give serial interfaces amid microcontroller & various other peripherals such as parallel port.

Timers: A microcontroller may be in-built with one or more timer or counters. The timers & counters control all counting & timing operations within a microcontroller. Timers are employed to count external pulses. The main operations performed by timers, are pulse generations, clock functions, frequency measuring, modulations, making oscillations, etc.

ADC: (Analog to digital converter) ADC is employed to convert analog signals to digital ones. The input signals need to be analog for ADC. The digital signal production can be employed for different digital applications (such as- measurement gadgets).

DAC: (digital to analog converter) this converter executes opposite functions that ADC perform. This device is generally employed to supervise analog appliances like- DC motors, etc.

Interpret Control: This controller is employed for giving delayed control for a working program. The interpret can be internal or external.

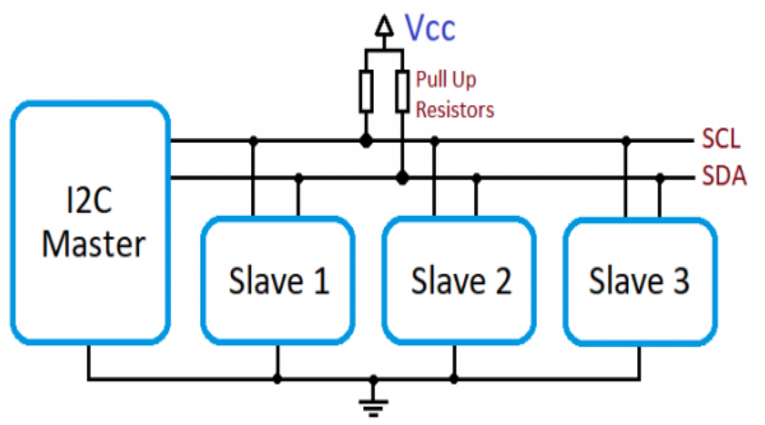
Special Functioning Block: Some special microcontrollers manufactured for special appliances like- space systems, robots, etc, comprise of this special function block. This special block has additional ports, so as to carry out some special operations.

1.4 The I2C Serial Communication Bus

I²C or I2C is an abbreviation of Inter-Integrated Circuit, a serial communication protocol made by Philips Semiconductor. It is created with an intention of communication between chips residing on the same Printed Circuit Board (PCB). It is commonly usually used to interface slow speed ICs to a microprocessor or a microcontroller. It is a master-slave protocol, usually a processor or a microcontroller is the master and other chips, for example Temperature Sensor, etc. will be the slave. We can have multiple masters and multiple slaves in the same I2C bus. Hence it is a multi-master, multi-slave protocol.

It needs only two wires for exchanging data and ground as the reference.

- SDA – Serial Data
- SCL – Serial Clock
- GND – Ground



I2C bus is popular because it is simple to use, there can be more than one master, only upper bus speed is defined and only two wires with pull-up resistors are needed to connect almost unlimited number of I2C devices.

Devices on an I2C bus are always a master or a slave.

Master is the device which always initiates a communication and drives the clock line (SCL). Usually, a microcontroller or microprocessor acts a master which needs to read data from or write data to slave peripherals.

A slave device is always responding to a master and won't initiate any communication by itself. Each slave device will have a unique address such that master can request data from or write data to it.

Each slave device has a unique address. Transfer from and to master device is serial and it is split into 8-bit packets. All these simple requirements make it very simple to implement I2C interface even with cheap microcontrollers that have no special I2C hardware controller. We only need 2 free I/O pins and few simple i2C routines to send and receive commands.

The initial I2C specifications defined maximum clock frequency of 100 kHz. This was later increased to 400 kHz as Fast mode. There is also a High-speed mode which can go up to 3.4 MHz and there is also a 5 MHz ultra-fast mode.

I2C Addresses

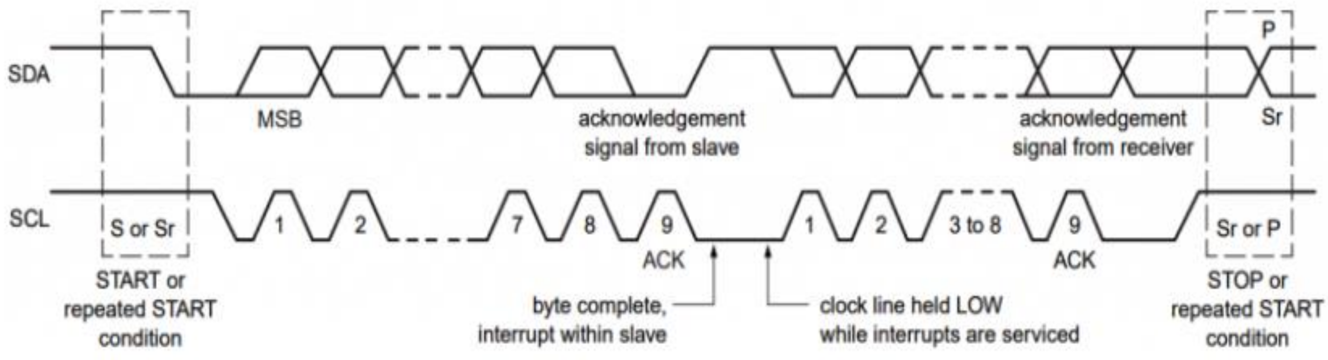
Basic I2C communication is using packets of 8 bits or bytes. Each I2C slave device has a 7-bit address that needs to be unique on the bus. Some devices have fixed I2C address while others have few address lines which determine lower bits of the I2C address. This makes it very easy to have all I2C devices on the bus with unique I2C address. There are also devices which have 10-bit address as allowed by the specification.

7-bit address represents bits 7 to 1 while bit 0 is used to signal reading from or writing to the device. If bit 0 (in the address byte) is set to 1 then the master device will read from the slave I2C device.

Master device needs no address since it generates the clock (via SCL) and addresses individual I2C slave devices.

I2C Protocol

I2C protocol is using only 2 lines (one for clock and one for data) for communication. But usually, we do not need to worry about it as in most of the device's hardware itself will take care of these things.


SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
P
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
byte complete, interrupt within slave
clock line held LOW while interrupts are serviced
STOP or repeated START condition

Start Condition

I2C start condition is issued by a master device to give a notice to all slave devices that the communication is about to start. Thus, start condition triggers all slave devices to listen to the data in the bus. To issue start condition, the master device pulls SDA low and leaves SCL high. In the case of multi-master I2C there is a possibility that 2 masters wish to take ownership of the bus at the same time. In these cases, the device which pull down SDA first gains the control of the bus.

Address Frame

Address frame is always sent just after the first start condition during every communication sequence. In this master devices specifies the address of the slave device to which the master wants to communicate. There are basically 2 types of addressing 7-bit addressing and 10-bit addressing. In the 7-bit addressing mode, master sends address first (MSB first) followed by read/write (R/W) indicating bit (0 => Write, 1 => Read).

Data Frames

Data frame(s) are transmitted just after the address frame. It can be sent from master to slave OR from slave to master depending on the above R/W bit through SDA line. The master will continue generating required clock signals. Devices can send one or more than one data frame as per the requirements.

Stop Condition

Master device will generate stop condition once all data frames has been sent/received. As per I2C standards, STOP condition is defined as a LOW to HIGH transition on SDA line after a LOW to HIGH transition on SCL, with SCL HIGH.

Acknowledge (ACK) and Not Acknowledge (NACK)

Each byte of data in I2C communication includes an additional bit known as ACK bit. This bit provides a provision for the receiver to send a signal to transmitter that the byte was successfully received and ready to accept another byte.

10-bit Addresses

We know that I2C bus uses 7-bit addressing, which means that devices are limited to 127 devices and address clashes can happen. 10-bit address scheme is introduced to solve this problem. 10-bit address devices can be mixed with 7-bit devices and it increases the address range about 10 times.

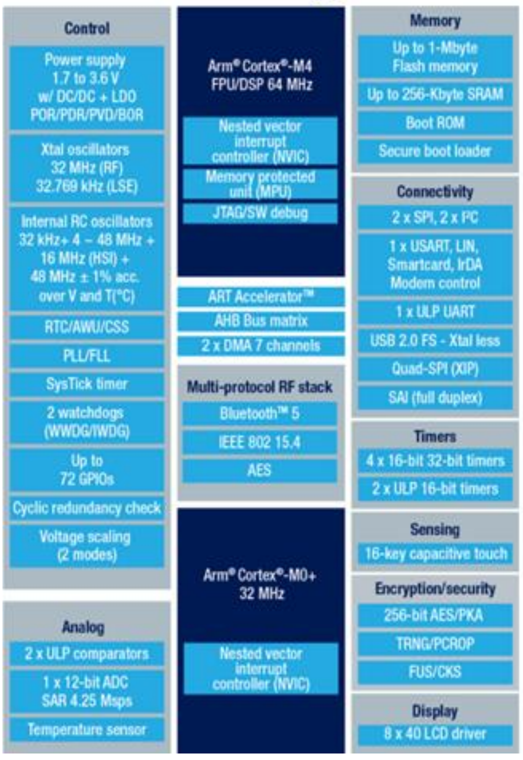
Chapter 2: Hardware used in the project

2.1 STM32WB55 Nucleo

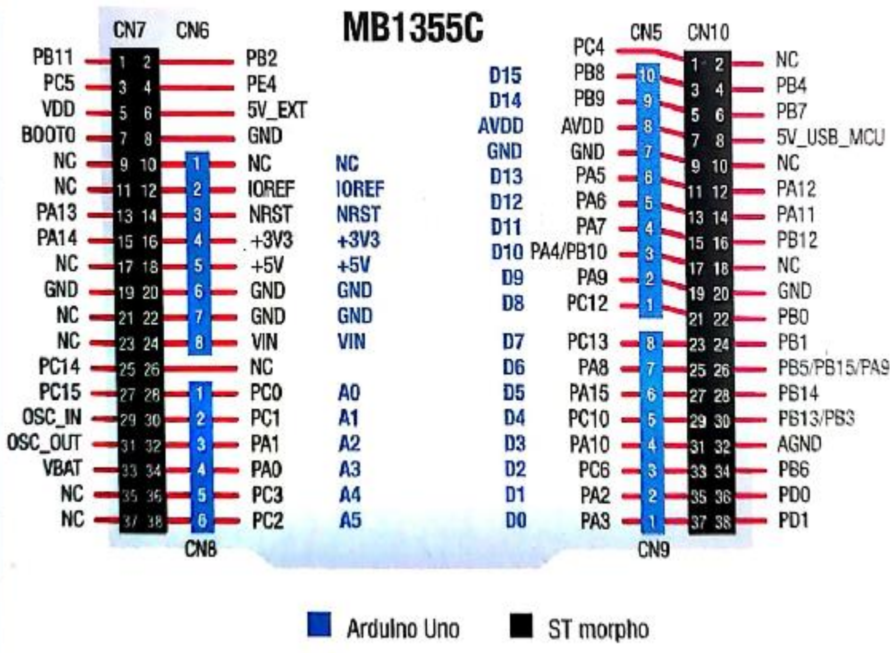
STM32 is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. The STM32 chips are grouped into related series that are based around the same 32-bit ARM processor core. Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various peripherals.
The unparalleled range of STM32 microcontrollers, based on an industry-standard core, comes with a vast choice of tools and software to support project development, making this family of products ideal for both small projects and end-to-end platforms.
All Nucleo boards by STMicroelectronics support the mbed IDE development and has an additional onboard ST-LINK/V2-1 host adapter chip that supplies SWD debugging, virtual COM port, mass storage. There are three Nucleo board families, each supporting a different microcontroller IC package footprint. The debugger embedded on Nucleo boards can be converted to SEGGER J-Link debugger protocol.

The STM32WB55xx and STM32WB35xx multiprotocol wireless and ultra-low-power devices embed a powerful and ultra-low-power radio compliant with the Bluetooth Low Energy SIG specification v5.0 and with IEEE 802.15.4-2011. They contain a dedicated Arm Cortex -M0+ for performing all the real-time low layer operation.
The devices are designed to be extremely low-power and are based on the high-performance Arm Cortex -M4 32-bit RISC core operating at a frequency of up to 64 MHz. The Cortex -M4 core features a Floating-point unit (FPU) single precision that supports all Arm single-precision data-processing instructions and data types. It also implements a full set of DSP instructions and a memory protection unit (MPU) that enhances application security.

Circuit Diagram



Stm32wb55 Pinout

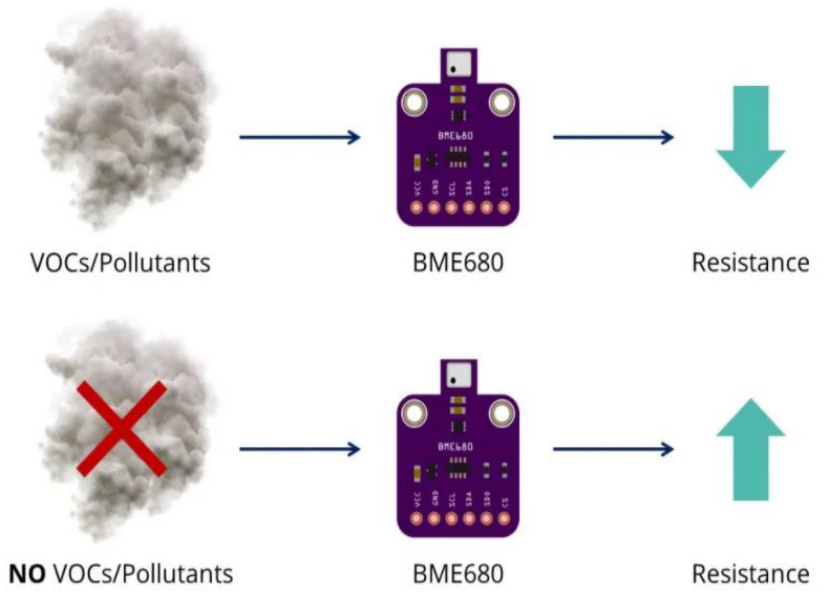


2.2 BME680

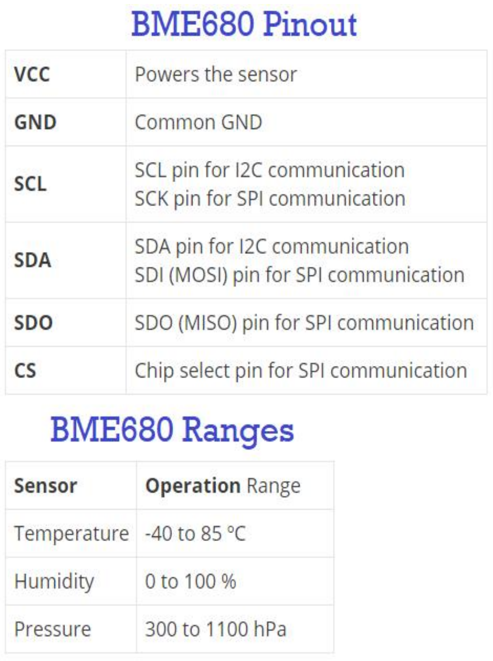
The BME680 is a digital 4-in-1 sensor from Bosch with gas, humidity, pressure, and temperature measurement based on proven sensing principles. The sensor module is housed in an extremely compact metal-lid LGA package with a footprint of only 3.0 × 3.0 mm² with a maximum height of 1.00 mm (0.93 ± 0.07 mm). Its small dimensions and its low power consumption enable the integration in battery-powered devices, such as handsets or wearables.

The sensor communicates with a microcontroller using I2C or SPI communication protocols. The gas sensor can detect a broad range of gases like volatile organic compounds (VOC). For this reason, the BME680 can be used in indoor air quality control.

The BME680 contains a MOX (Metal-oxide) sensor that detects VOCs in the air. This sensor gives you a qualitative idea of the sum of VOCs/contaminants in the surrounding air – it is not specific for a specific gas molecule.
MOX sensors are composed of a metal-oxide surface, a sensing chip to measure changes in conductivity, and a heater. It detects VOCs by adsorption of oxygen molecules on its sensitive layer. When the sensor comes into contact with the reducing gases, the oxygen molecules react and increase the conductivity across the surface. As a raw signal, the BME680 outputs resistance values. These values change due to variations in VOC concentrations:



- **Higher** concentration of VOCs » **Lower** resistance
- **Lower** concentration of VOCs » **Higher** resistance



BME680 Pinout

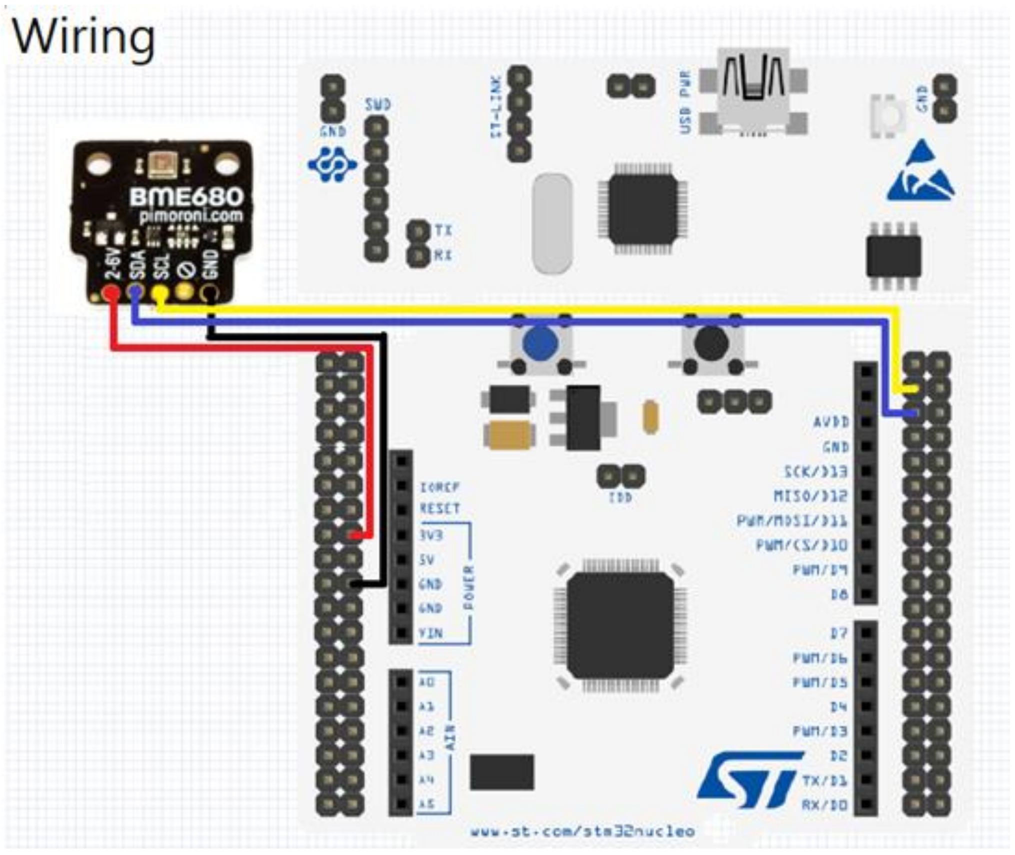
VCC	Powers the sensor
GND	Common GND
SCL	SCL pin for I2C communication SCK pin for SPI communication
SDA	SDA pin for I2C communication SDI (MOSI) pin for SPI communication
SDO	SDO (MISO) pin for SPI communication
CS	Chip select pin for SPI communication

BME680 Ranges

Sensor	**Operation** Range
Temperature	-40 to 85 ºC
Humidity	0 to 100 %
Pressure	300 to 1100 hPa

Chapter 3: Hardware Setup

3.1 STM32-BME680 Connection Diagram



STM32	BME680
pin 16 of CH7 (left side)	Vcc
pin 20 of CH7 (left side)	Gnd
pin 5 of CH10 (right side)	SDA
pin 3 of CH10 (right side)	SCL

We can power the Stm32WB55 from the pin 24 of CH7(left side) or from the mini-usb port of ST-Link.

The STM32 communicates with BME680 via a I2C bus.

- Vcc=2-6V for BME680 (in this project we have used 3.3V)

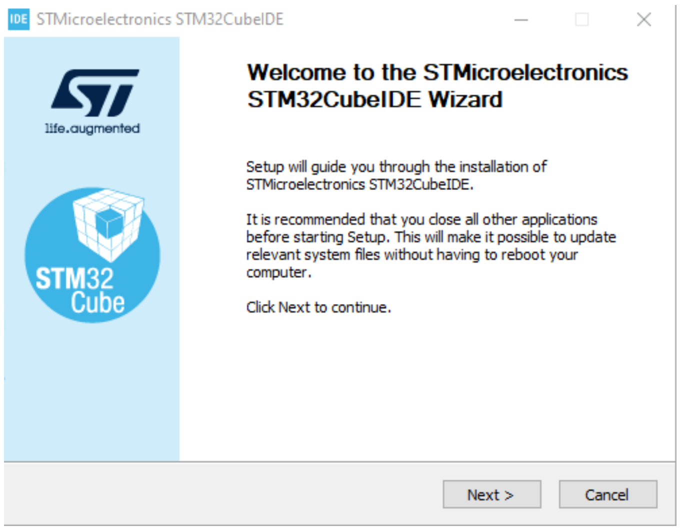
(For debugging reasons, we can use an usb to serial convertor to take a serial output. It is not a necessary part of the project)

Chapter 4: Software Tools

4.1 Installation of STM32 Cube IDE

The main tool that we will use is the STM32 CubeIDE which is an eclipse-based IDE. To download the tool, simply, visit the site (https://www.st.com/en/development-tools/stm32cubeide.html?ecmp=tt11319_gl_link_may2019&2).

To install, simply follow the installation Wizard.


STMicroelectronics STM32CubeIDE

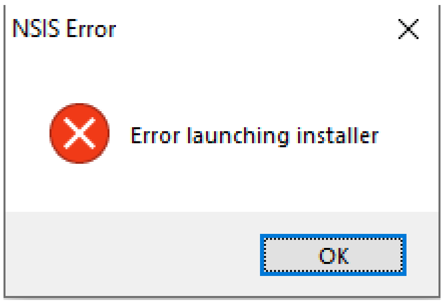
Welcome to the STMicroelectronics STM32CubeIDE Wizard

Setup will guide you through the installation of STMicroelectronics STM32CubeIDE.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

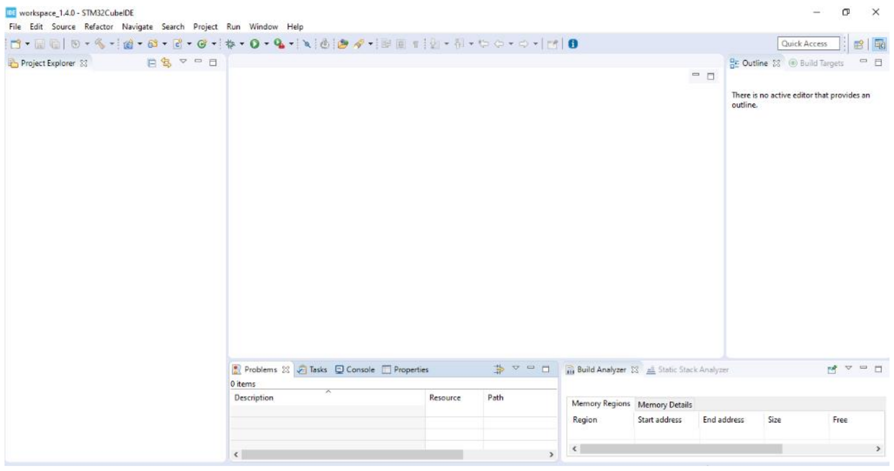
Click Next to continue.

If you run into an error like the following, replace the .exe file to your c:/


NSIS Error

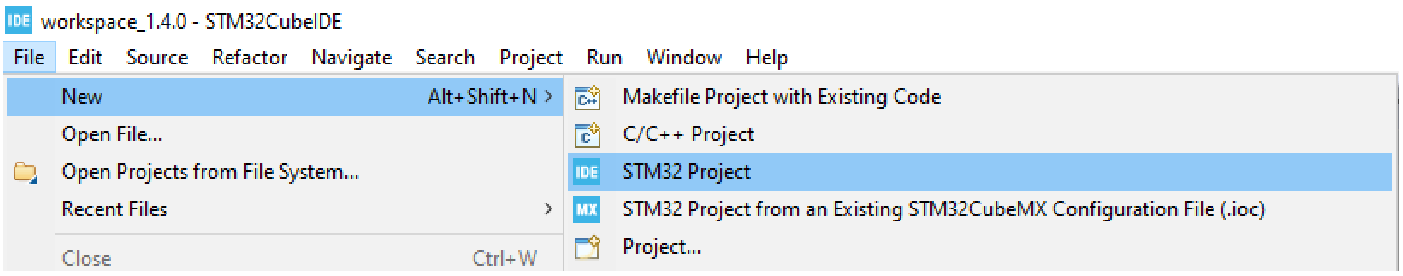
Error launching installer

Install all the necessary drivers when asked. Once the installation is completed and execute the tool, we will see the following screen.

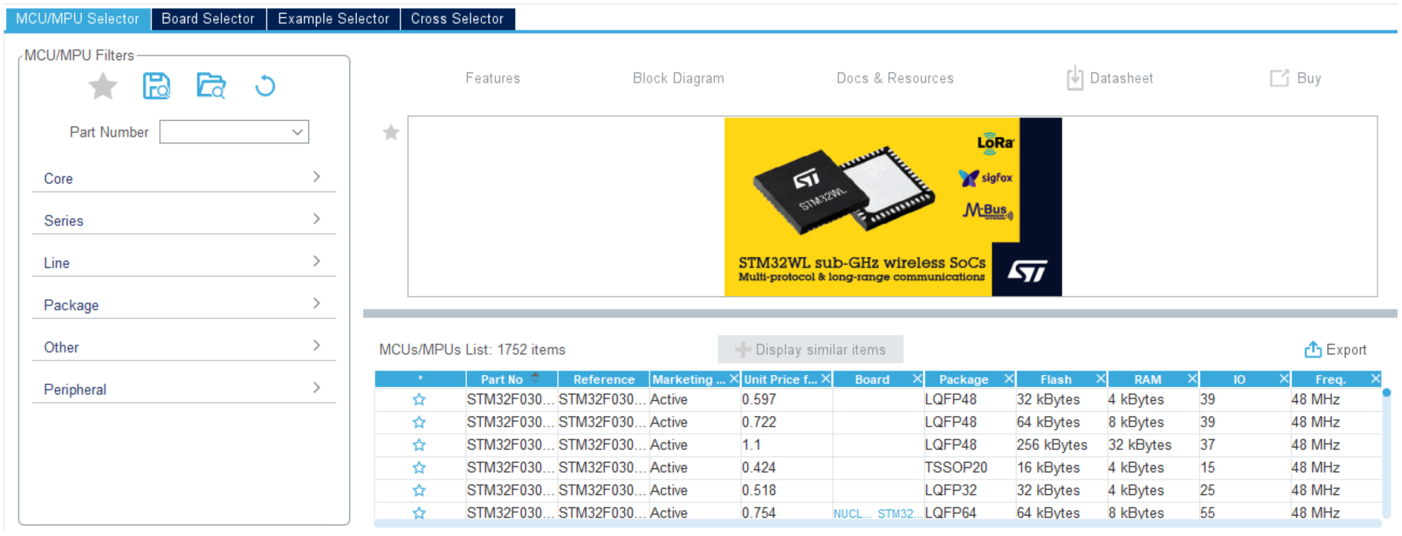

workspace_1.4.0 - STM32CubeIDE

There is no active editor that provides an outline.

Then select File -> New -> STM32 Project


workspace_1.4.0 - STM32CubeIDE
File
Edit
Source
Refactor
Navigate
Search
Project
Run
Window
Help
New Alt+Shift+N >
Open File...
Open Projects from File System...
Recent Files >
Close Ctrl+W
Makefile Project with Existing Code
C/C++ Project
STM32 Project
STM32 Project from an Existing STM32CubeMX Configuration File (.ioc)
Project...

The IDE will start downloading the necessary libraries to display. Wait for a few minutes and then you will see the following image:

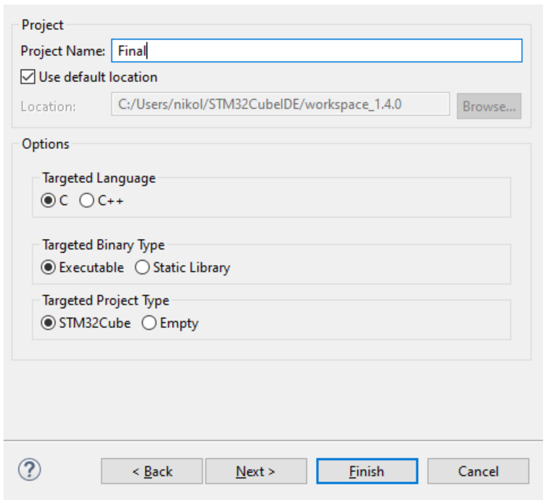


MCUs/MPUs List: 1752 items

*	Part No	Reference	Marketing ...	Unit Price f...	Board	Package	Flash	RAM	IO	Freq.
☆	STM32F030...	STM32F030...	Active	0.597		LQFP48	32 kBytes	4 kBytes	39	48 MHz
☆	STM32F030...	STM32F030...	Active	0.722		LQFP48	64 kBytes	8 kBytes	39	48 MHz
☆	STM32F030...	STM32F030...	Active	1.1		LQFP48	256 kBytes	32 kBytes	37	48 MHz
☆	STM32F030...	STM32F030...	Active	0.424		TSSOP20	16 kBytes	4 kBytes	15	48 MHz
☆	STM32F030...	STM32F030...	Active	0.518		LQFP32	32 kBytes	4 kBytes	25	48 MHz
☆	STM32F030...	STM32F030...	Active	0.754	NUCL... STM32...	LQFP64	64 kBytes	8 kBytes	55	48 MHz

Here we are called to complete the type of the stm32 device we are about to use. In our case we complete the STM32WB55RG and we select and click next on the result with the nucleo mark.

We give a name to our project (in our case is Final) and choose the language we will use (we will use C). Leave the rest of the choices as it is.



Project

Project Name: Final

☑ Use default location

Location: C:/Users/nikol/STM32CubeIDE/workspace_1.4.0

Options

Targeted Language: [✓ C] C++

Targeted Binary Type: [✓ Executable] Static Library

Targeted Project Type: [✓ STM32Cube] Empty

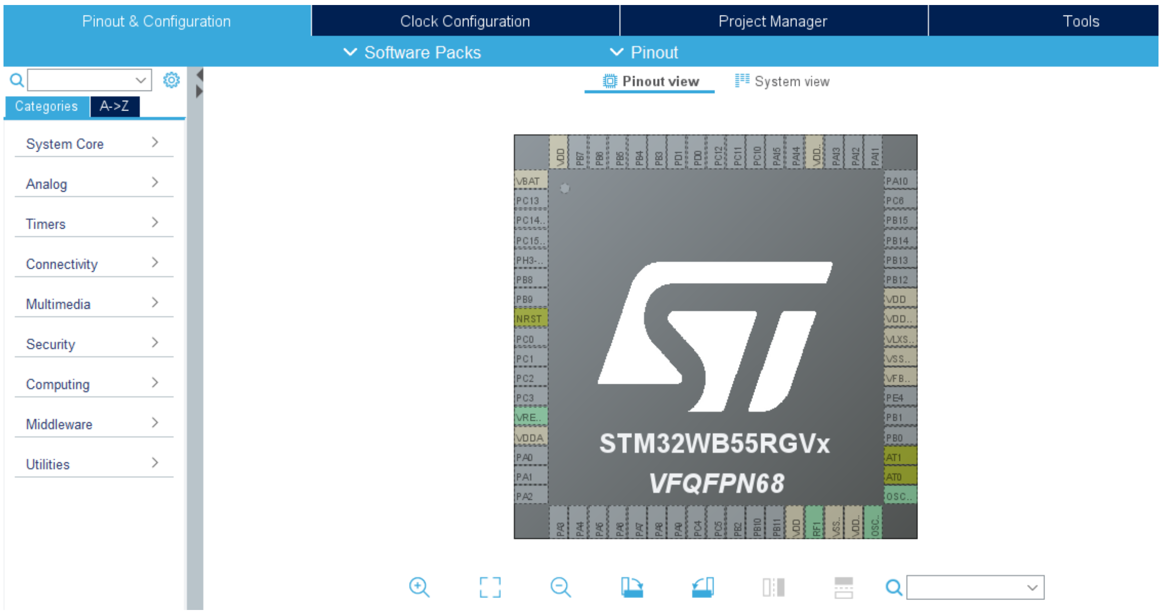
The project will start building the necessary code and visual display and will take some time.

When it is completed, we are ready to start working on our project.

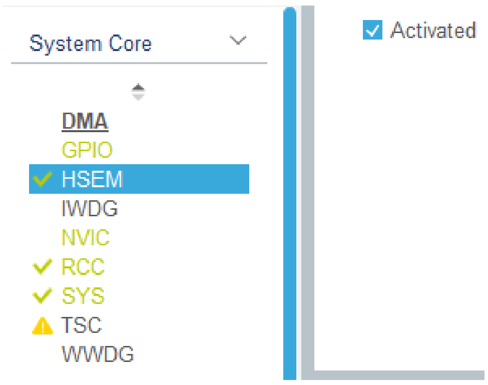
Chapter 5: Software Code and Settings

5.1 Device Configuration Tool

At first, we are looking at an empty project with our specific stm32.

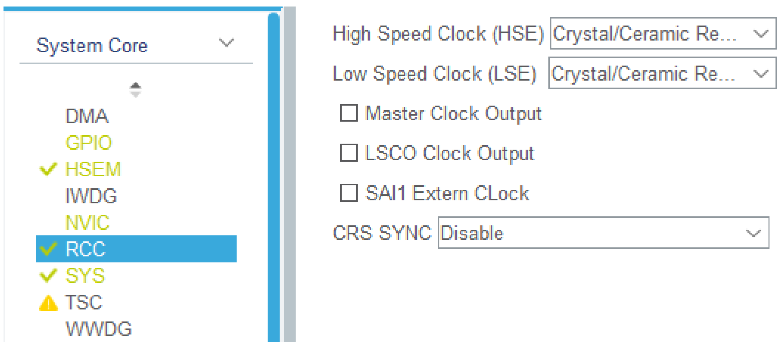


1. Here we can change the hardware setting of the device, either from the left list or by directly pressing on each pin.
   We begin with the System Core -> HSEM -> tick on activation. We need this part for our BLE libraries.



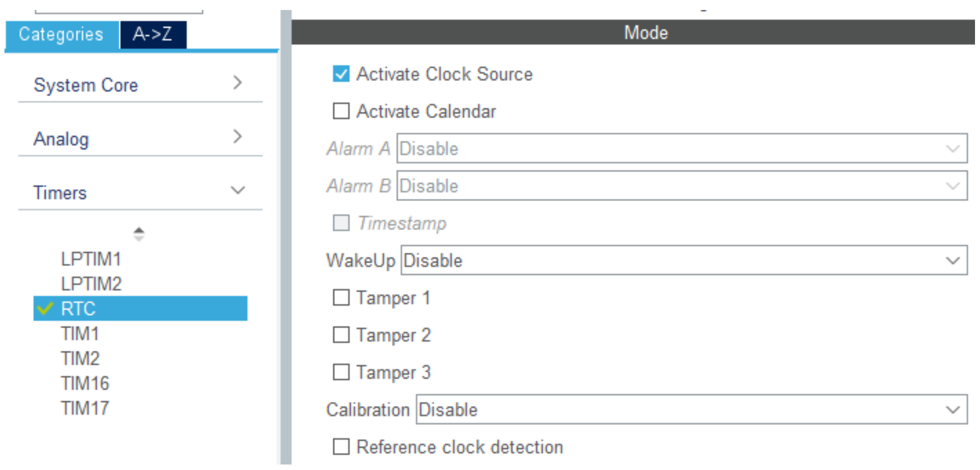
☑ Activated

2. Next, we choose the RCC which we use to give timing to our BLE. We make the changes:
   We change both clocks (low and high) to the Crystal/Ceramic Resonator.
   This will activate pins PC14, PC15 accordingly.



High Speed Clock (HSE) Crystal/Ceramic Re...
Low Speed Clock (LSE) Crystal/Ceramic Re...
☐ Master Clock Output
☐ LSCO Clock Output
☐ SAI1 Extern CLock
CRS SYNC Disable

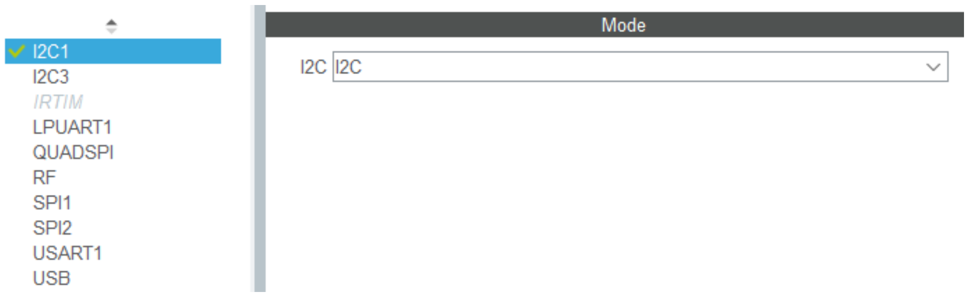
3. After that, we go to the Timers, where we need to activate the RTC.



☑ Activate Clock Source
☐ Activate Calendar
Alarm A Disable
Alarm B Disable
☐ Timestamp
WakeUp Disable
☐ Tamper 1
☐ Tamper 2
☐ Tamper 3
Calibration Disable
☐ Reference clock detection

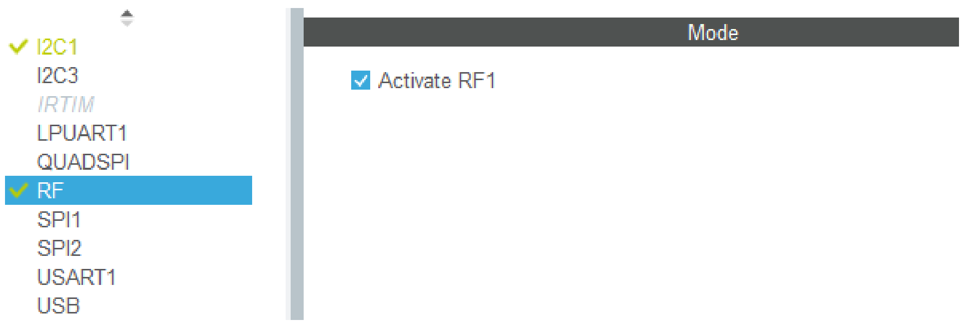
The RTC is used by the Middleware to manage the RF wakeup system.

4. Up next, we go to Connectivity and we need to activate I2C in order for the STM32 to communicate with the BME680. Pins PB8(SCL) and PB9(SDA) will be marked.



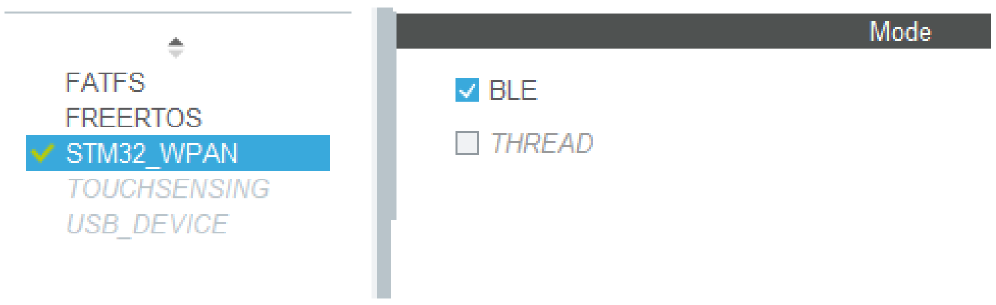
I2C I2C

5. Also, we will activate the RF for the BLE function. Pin RF1 will be highlighted.



☑ Activate RF1

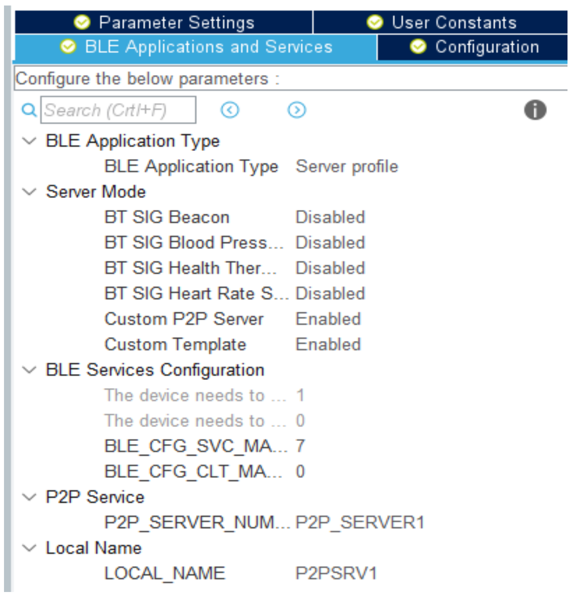
6. Finally, we will go to the Middleware and activate STM32_WPAN -> BLE to activate the BLE function.



☑ BLE
☐ THREAD

Also, we need to change the Configuration at the BLE Applications and Services Tab.

We must change the Custom Template to enable, to be able to send our own services and from the Local Name we can name our Bluetooth device. The final form must be:



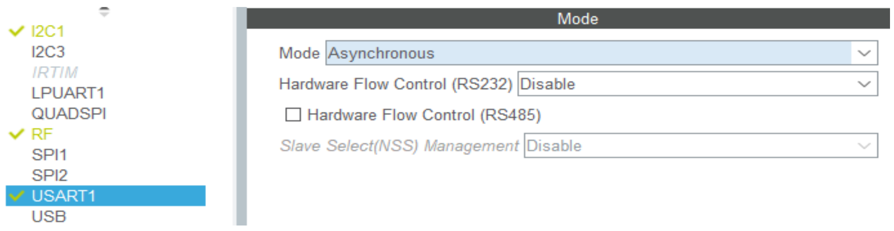
Configure the below parameters :

BLE Application Type	
BLE Application Type	Server profile
Server Mode	
BT SIG Beacon	Disabled
BT SIG Blood Press...	Disabled
BT SIG Health Ther...	Disabled
BT SIG Heart Rate S...	Disabled
Custom P2P Server	Enabled
Custom Template	Enabled
BLE Services Configuration	
The device needs to ...	1
The device needs to ...	0
BLE_CFG_SVC_MA...	7
BLE_CFG_CLT_MA...	0
P2P Service	
P2P_SERVER_NUM...	P2P_SERVER1
Local Name	
LOCAL_NAME	P2PSRV1

Now as an extra step that is not necessary for the final result we can activate the Usart1 in order to be able to get serial output for debugging purposes, and enable a led in order to test our device connection via Bluetooth.

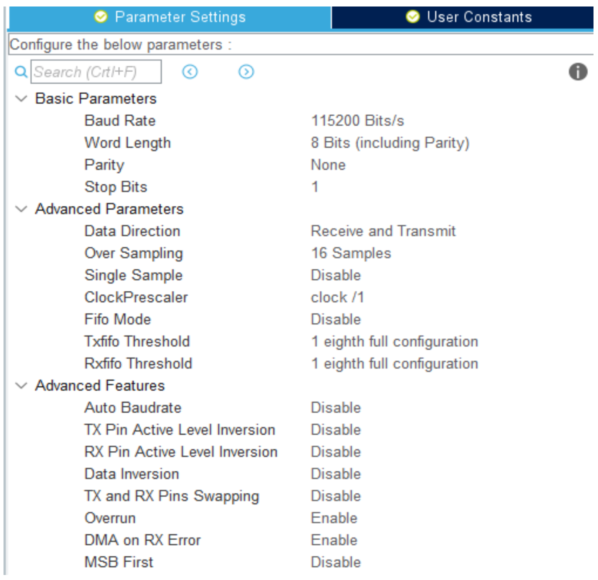
Steps 7,8 can be ignored.

7. We move to Connectivity->USART1 -> change the mode to Asynchronous



Mode: Asynchronous
Hardware Flow Control (RS232): Disable
☐ Hardware Flow Control (RS485)
Slave Select(NSS) Management: Disable

Also, at the configuration we can change the parameter Settings. In our case we have the default.

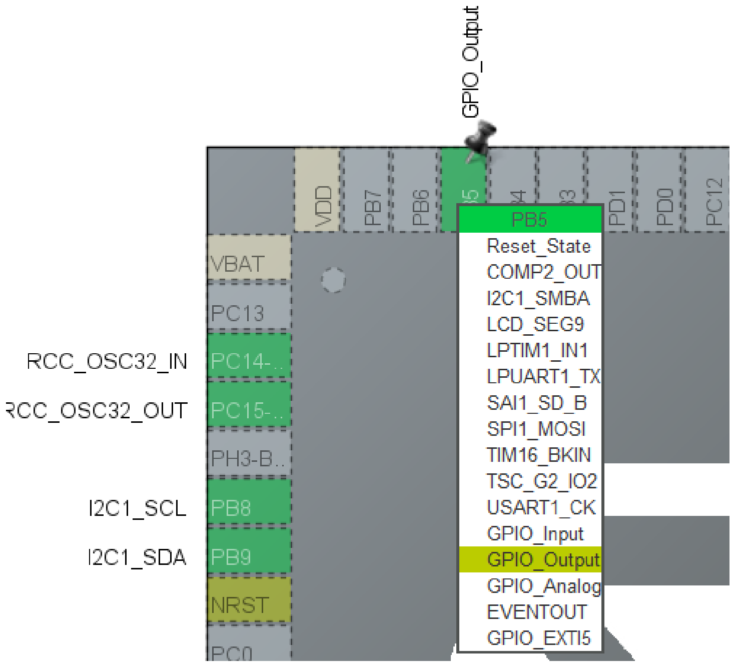


Configure the below parameters :

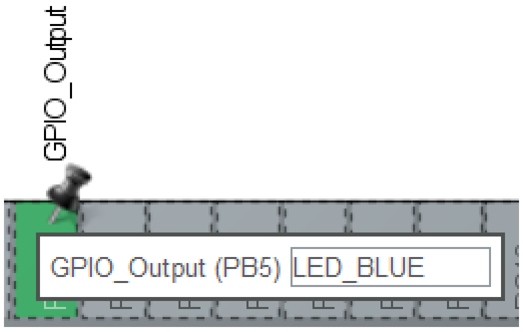
Basic Parameters	
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable
ClockPrescaler	clock /1
Fifo Mode	Disable
Txfifo Threshold	1 eighth full configuration
Rxfifo Threshold	1 eighth full configuration
Advanced Features	
Auto Baudrate	Disable
TX Pin Active Level Inversion	Disable
RX Pin Active Level Inversion	Disable
Data Inversion	Disable
TX and RX Pins Swapping	Disable
Overrun	Enable
DMA on RX Error	Enable
MSB First	Disable

Pins PA9(TX) and PA10(RX) will be highlighted. Note: To get a serial out an usb to serial convertor must be used.

8. For the Led we go manually on pin PB5 and select from the list GPIO_Output.

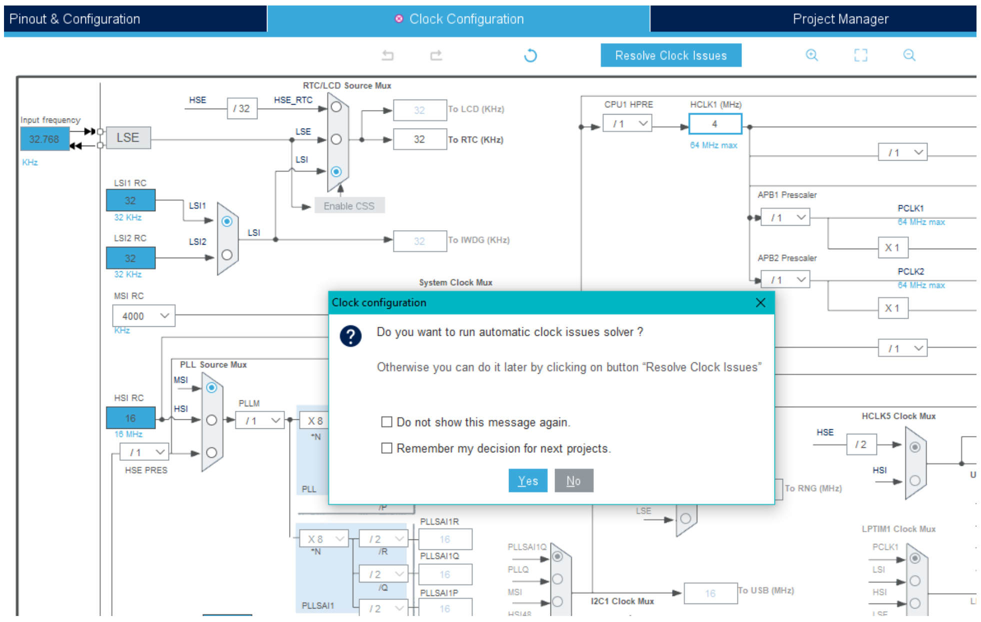


For easier use, we right click on the pin ->Enter User Label to name it. We will name it as LED_BLUE



GPIO_Output (PB5) LED_BLUE

All the pinout and Configurations are completed. Although we have completed everything, we can notice an error notification on Clock Configuration Tab. We move to this tab to fix the error. When the program will ask to solve the issue, we press yes.



Clock configuration

Do you want to run automatic clock issues solver ?

Otherwise you can do it later by clicking on button "Resolve Clock Issues"

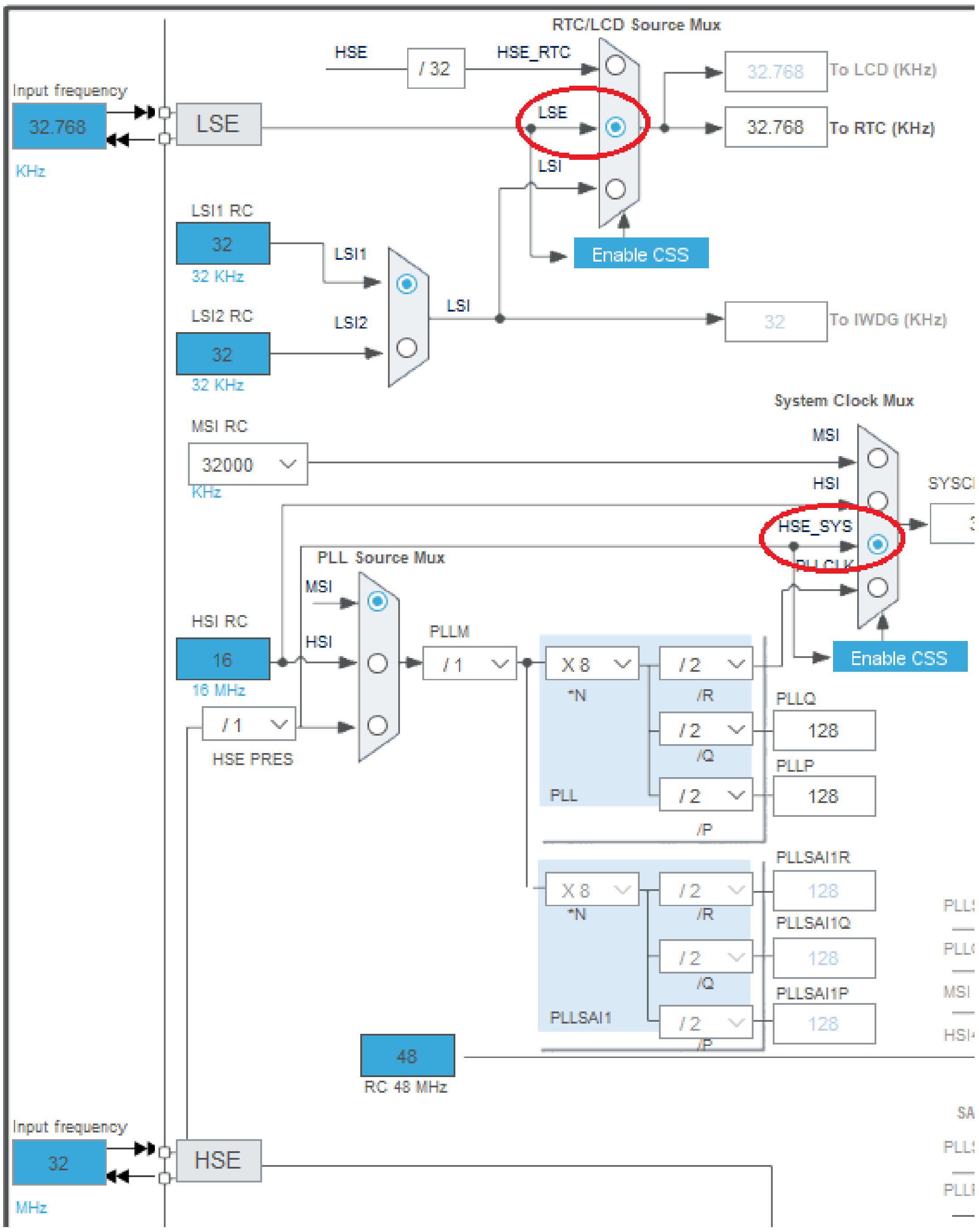
☐ Do not show this message again.

☐ Remember my decision for next projects.

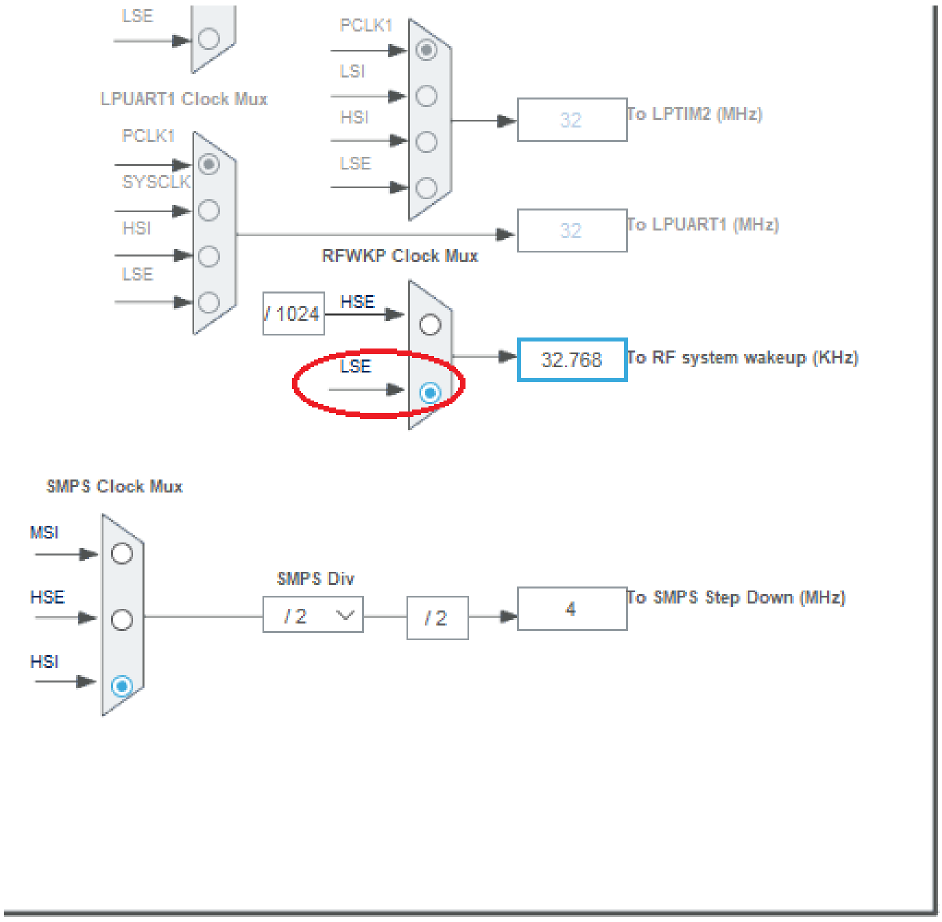
Yes No

This action will solve partially the issue. Next, we must make some changes. In the top left corner, we enabled our low-speed external crystal (32.768 KHZ). So, we must make sure that this is selected in our multiplexer (Mux). We change the Mux to LSE.

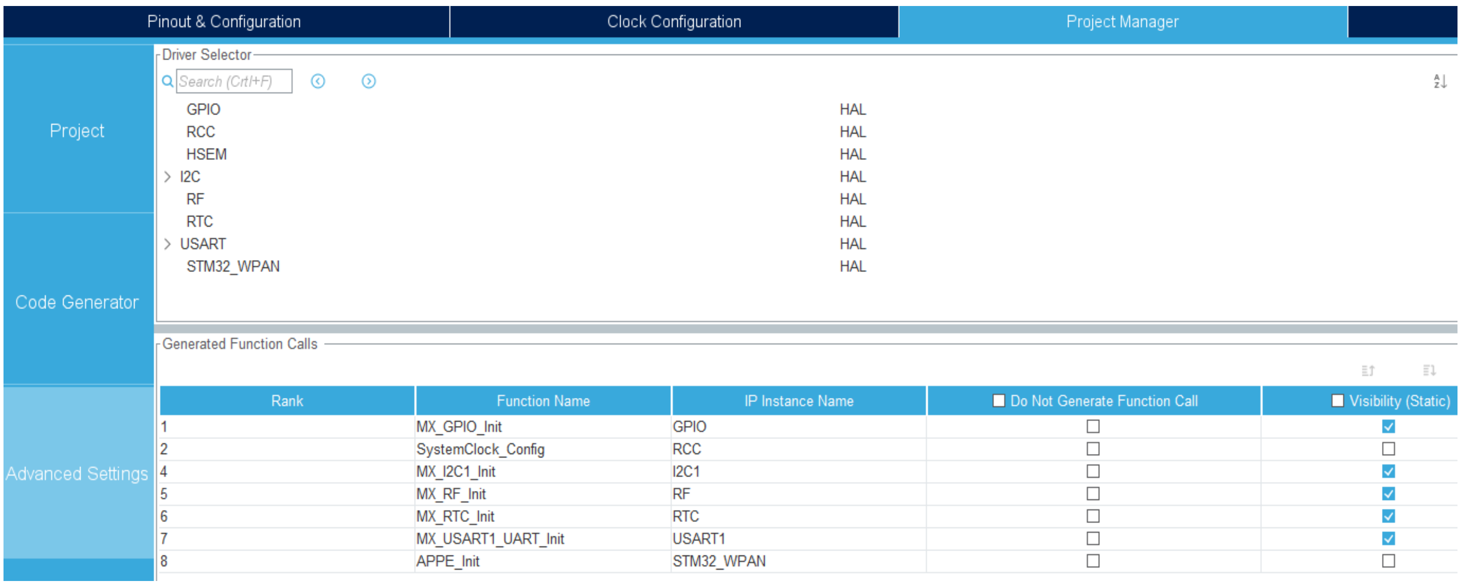
Next, we enabled our high-speed external crystal (32 MHZ). Also, here we must change the multiplexer, this time to HSE_SYS.



Finally, we move to the bottom right corner we have the RF system wakeup and again we need to change to multiplexer to LSE



The last step is to choose the Project manager tab -> Advanced Settings and then check the Generated Function Calls. All peripherals must be enabled before we enable the middleware.



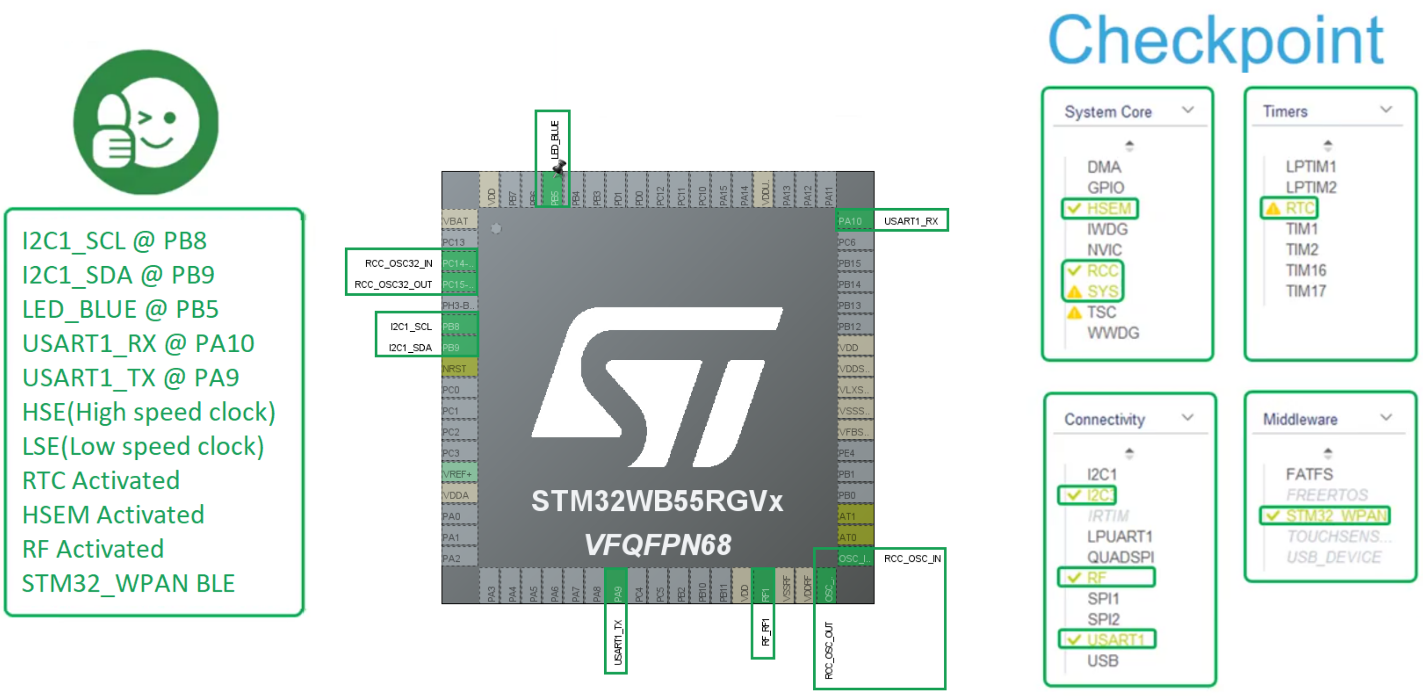
Driver Selector

GPIO	HAL
RCC	HAL
HSEM	HAL
> I2C	HAL
RF	HAL
RTC	HAL
> USART	HAL
STM32_WPAN	HAL

Generated Function Calls

Rank	Function Name	IP Instance Name	Do Not Generate Function Call	Visibility (Static)
1	MX_GPIO_Init	GPIO	☐	✓
2	SystemClock_Config	RCC	☐	☐
4	MX_I2C1_Init	I2C1	☐	✓
5	MX_RF_Init	RF	☐	✓
6	MX_RTC_Init	RTC	☐	✓
7	MX_USART1_UART_Init	USART1	☐	✓
8	APPE_Init	STM32_WPAN	☐	☐

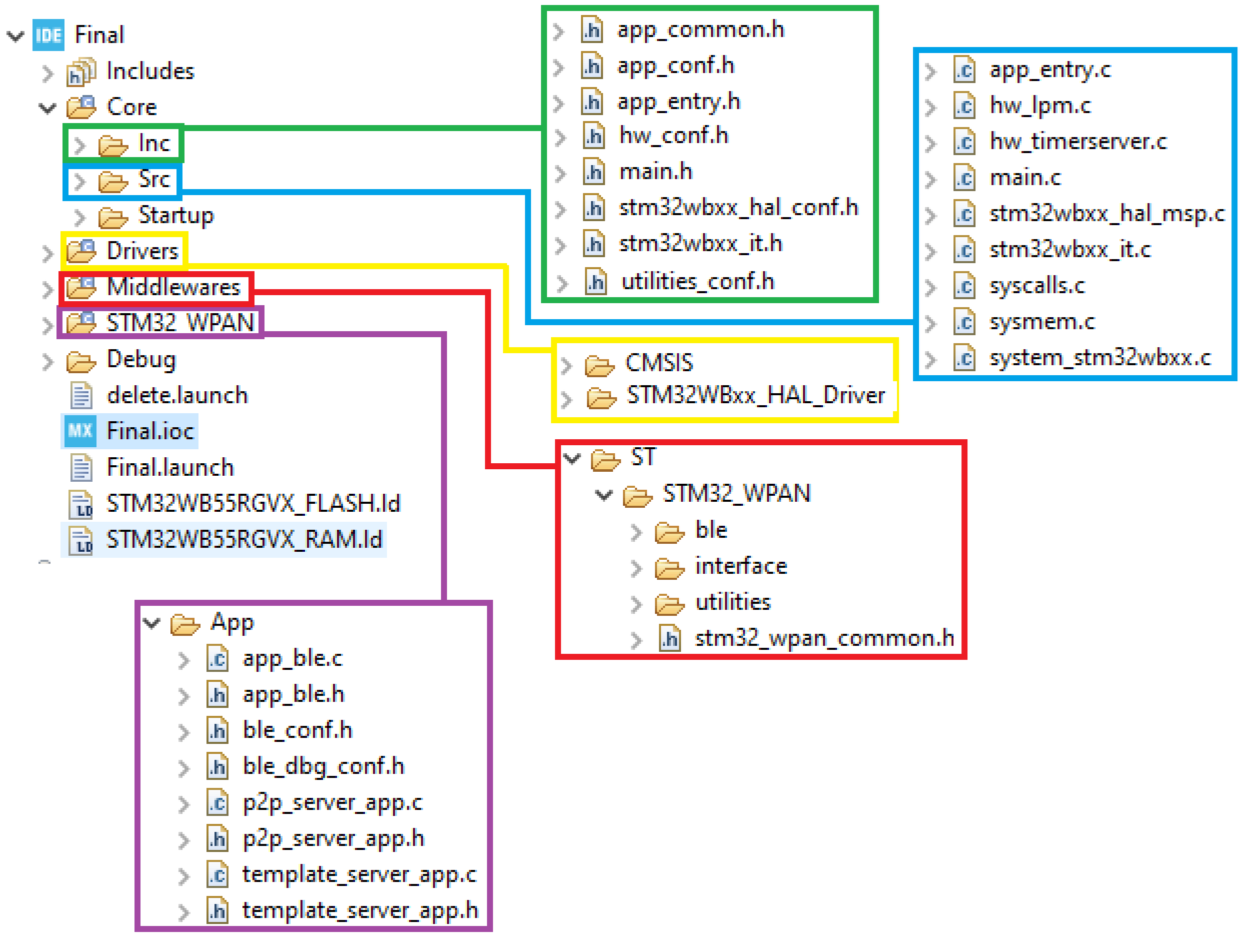
In Conclusion, this must be the final settings.



I2C1_SCL @ PB8
I2C1_SDA @ PB9
LED_BLUE @ PB5
USART1_RX @ PA10
USART1_TX @ PA9
HSE(High speed clock)
LSE(Low speed clock)
RTC Activated
HSEM Activated
RF Activated
STM32_WPAN BLE

If everything is completed, press save, and the program will generate the necessary code.

The generated code project tree must be like the following image:



```
Final
  Includes
  Core
    Inc
    Src
    Startup
  Drivers
  Middlewares
  STM32_WPAN
  Debug
  delete.launch
  Final.ioc
  Final.launch
  STM32WB55RGVX_FLASH.ld
  STM32WB55RGVX_RAM.ld
```

```
app_common.h
app_conf.h
app_entry.h
hw_conf.h
main.h
stm32wbxx_hal_conf.h
stm32wbxx_it.h
utilities_conf.h
```

```
app_entry.c
hw_lpm.c
hw_timerserver.c
main.c
stm32wbxx_hal_msp.c
stm32wbxx_it.c
syscalls.c
sysmem.c
system_stm32wbxx.c
```

```
CMSIS
STM32WBxx_HAL_Driver
```

```
ST
  STM32_WPAN
    ble
    interface
    utilities
    stm32_wpan_common.h
```

```
App
  app_ble.c
  app_ble.h
  ble_conf.h
  ble_dbg_conf.h
  p2p_server_app.c
  p2p_server_app.h
  template_server_app.c
  template_server_app.h
```

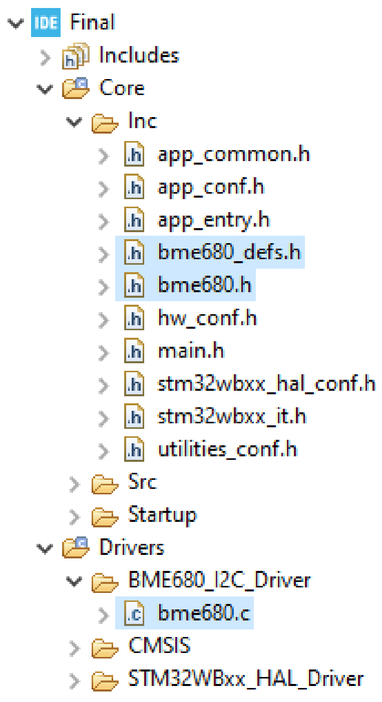
5.2 BME680 Code

The first step to start writing the code is to add the necessary libraries for the BME680.

We can download the necessary files from (https://github.com/BoschSensortec/BME680_driver).

Then create a new file inside the Drivers folder and place inside the **bme680.c**

Also, add the files **bme680_defs.h** and **bme680.h** inside the Inc file.

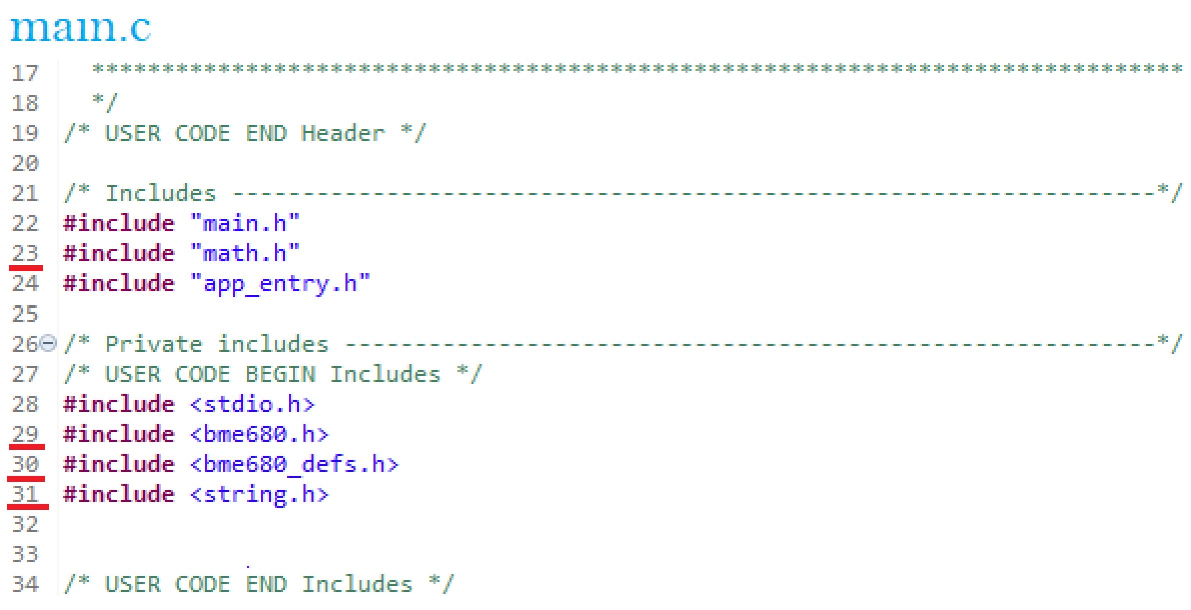


```
Final
  Includes
  Core
    Inc
      app_common.h
      app_conf.h
      app_entry.h
      bme680_defs.h
      bme680.h
      hw_conf.h
      main.h
      stm32wbxx_hal_conf.h
      stm32wbxx_it.h
      utilities_conf.h
    Src
    Startup
  Drivers
    BME680_I2C_Driver
      bme680.c
    CMSIS
    STM32WBxx_HAL_Driver
```

Once we have all the necessary files added, we can proceed to the Src-> **main.c**

Starting, we need to add the libraries that we will use and the drivers of the bme680.

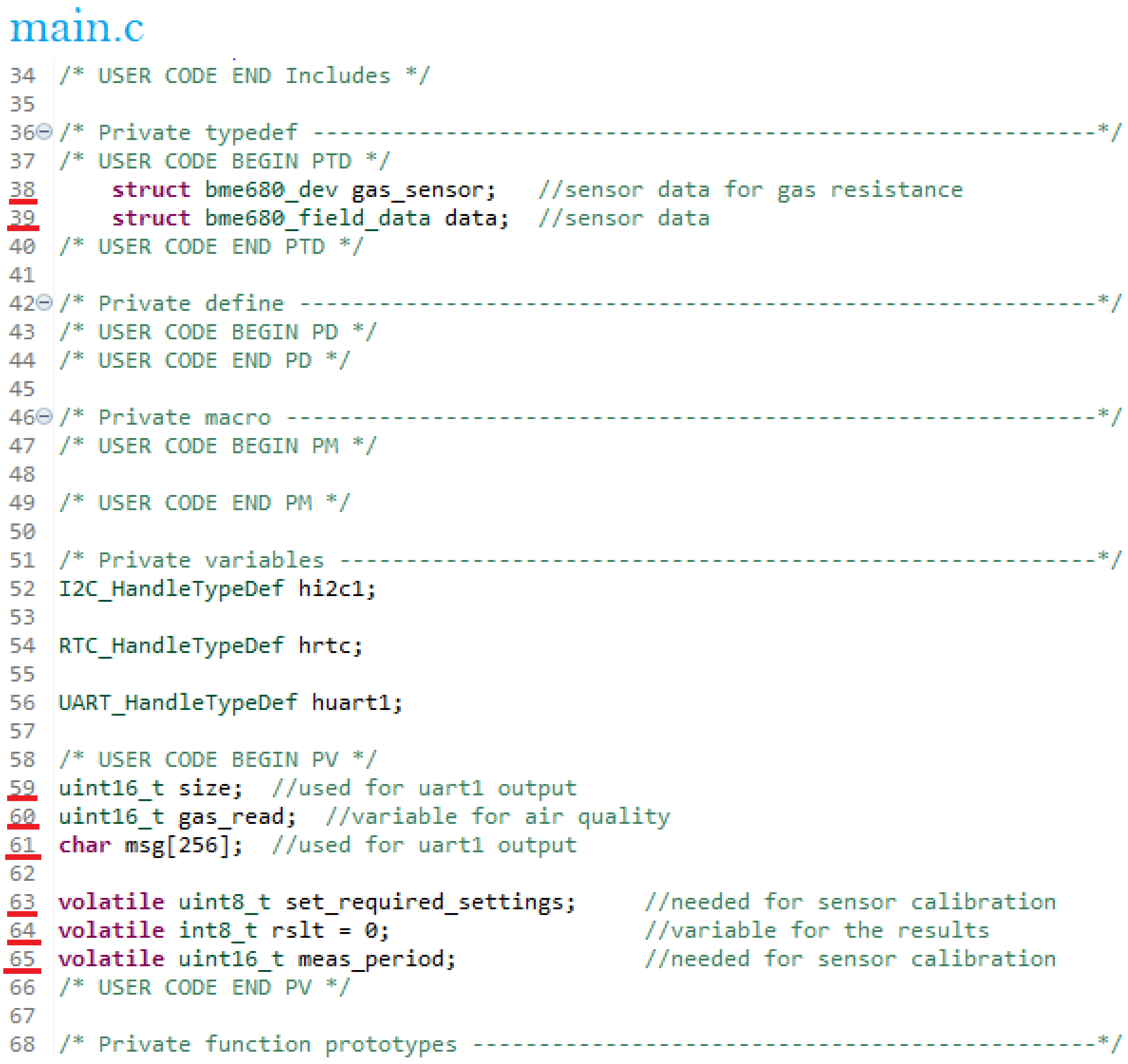
We begin at line 23,29, 30, 31. We added the **string.h** to be able to take an output at uart1 for debugging reasons and **math.h** to be able to use log.



main.c

```
17    ******************************************************************************
18    */
19  /* USER CODE END Header */
20
21  /* Includes ------------------------------------------------------------------*/
22  #include "main.h"
23  #include "math.h"
24  #include "app_entry.h"
25
26  /* Private includes ----------------------------------------------------------*/
27  /* USER CODE BEGIN Includes */
28  #include <stdio.h>
29  #include <bme680.h>
30  #include <bme680_defs.h>
31  #include <string.h>
32
33
34  /* USER CODE END Includes */
```

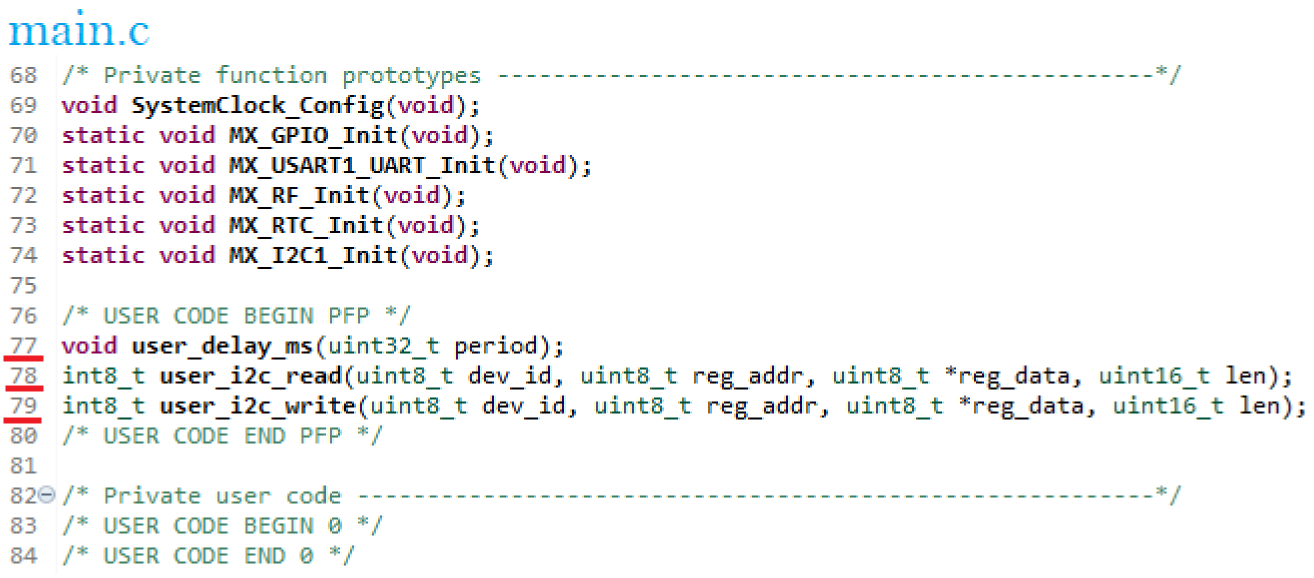
We move on by adding lines 38, 39, 59, 60, 61 ,63 ,64, 65 as shown. Here we declare the variables that we will use.



main.c

```c
34  /* USER CODE END Includes */
35
36⊖ /* Private typedef -----------------------------------------------------------*/
37  /* USER CODE BEGIN PTD */
38      struct bme680_dev gas_sensor;   //sensor data for gas resistance
39      struct bme680_field_data data;  //sensor data
40  /* USER CODE END PTD */
41
42⊖ /* Private define ------------------------------------------------------------*/
43  /* USER CODE BEGIN PD */
44  /* USER CODE END PD */
45
46⊖ /* Private macro -------------------------------------------------------------*/
47  /* USER CODE BEGIN PM */
48
49  /* USER CODE END PM */
50
51  /* Private variables ---------------------------------------------------------*/
52  I2C_HandleTypeDef hi2c1;
53
54  RTC_HandleTypeDef hrtc;
55
56  UART_HandleTypeDef huart1;
57
58  /* USER CODE BEGIN PV */
59  uint16_t size;  //used for uart1 output
60  uint16_t gas_read;  //variable for air quality
61  char msg[256];  //used for uart1 output
62
63  volatile uint8_t set_required_settings;     //needed for sensor calibration
64  volatile int8_t rslt = 0;                   //variable for the results
65  volatile uint16_t meas_period;              //needed for sensor calibration
66  /* USER CODE END PV */
67
68  /* Private function prototypes -----------------------------------------------*/
```

Finally, we add lines 77, 78, 79 as shown, to declare some essential functions for the sensor.

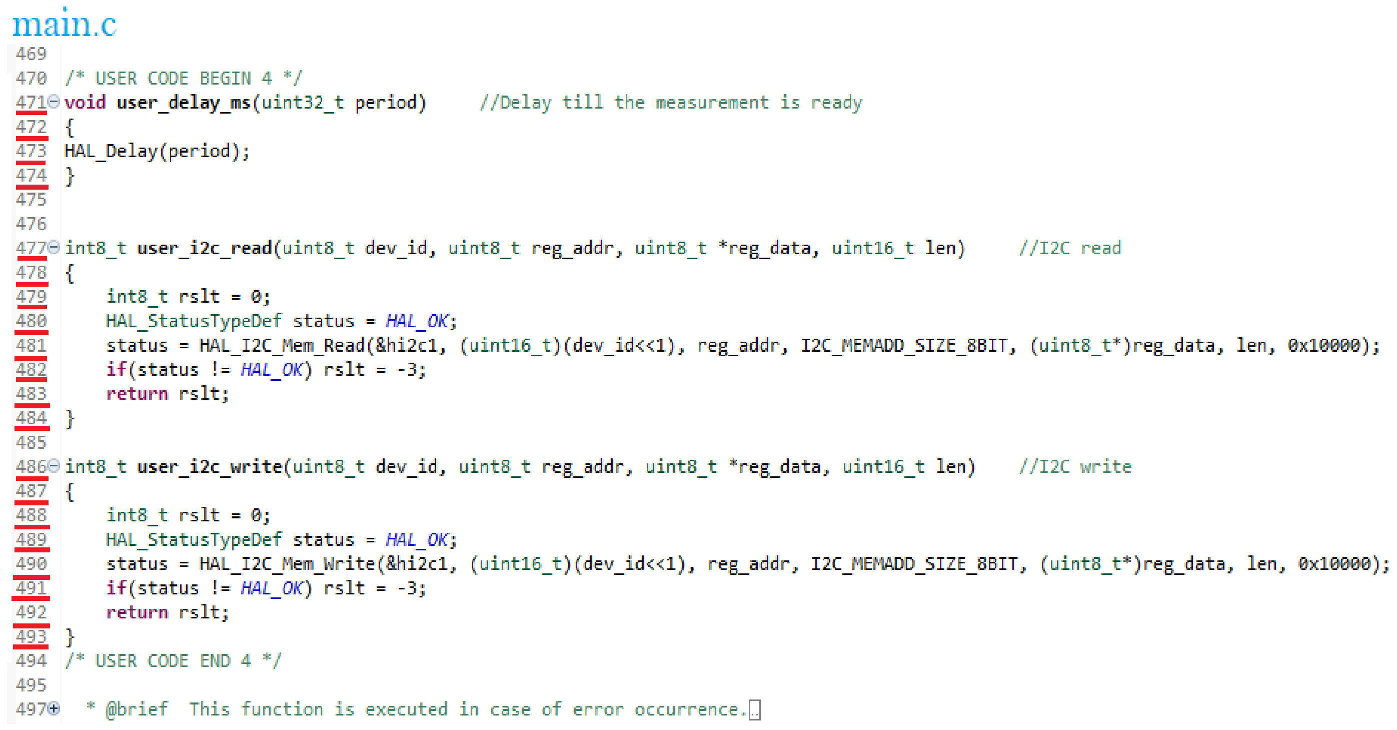


main.c

```c
68  /* Private function prototypes -----------------------------------------------*/
69  void SystemClock_Config(void);
70  static void MX_GPIO_Init(void);
71  static void MX_USART1_UART_Init(void);
72  static void MX_RF_Init(void);
73  static void MX_RTC_Init(void);
74  static void MX_I2C1_Init(void);
75
76  /* USER CODE BEGIN PFP */
77  void user_delay_ms(uint32_t period);
78  int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);
79  int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);
80  /* USER CODE END PFP */
81
82⊖ /* Private user code ---------------------------------------------------------*/
83  /* USER CODE BEGIN 0 */
84  /* USER CODE END 0 */
```

Since we have declared some function, we must also add them. So, we add lines 471 till 493 in the **“USER CODE BEGIN 4”** part as shown below. Those functions are necessary for the communication of the sensor with the microcontroller.

We also can find these functions at the **README.md** file that we have downloaded with the drivers.



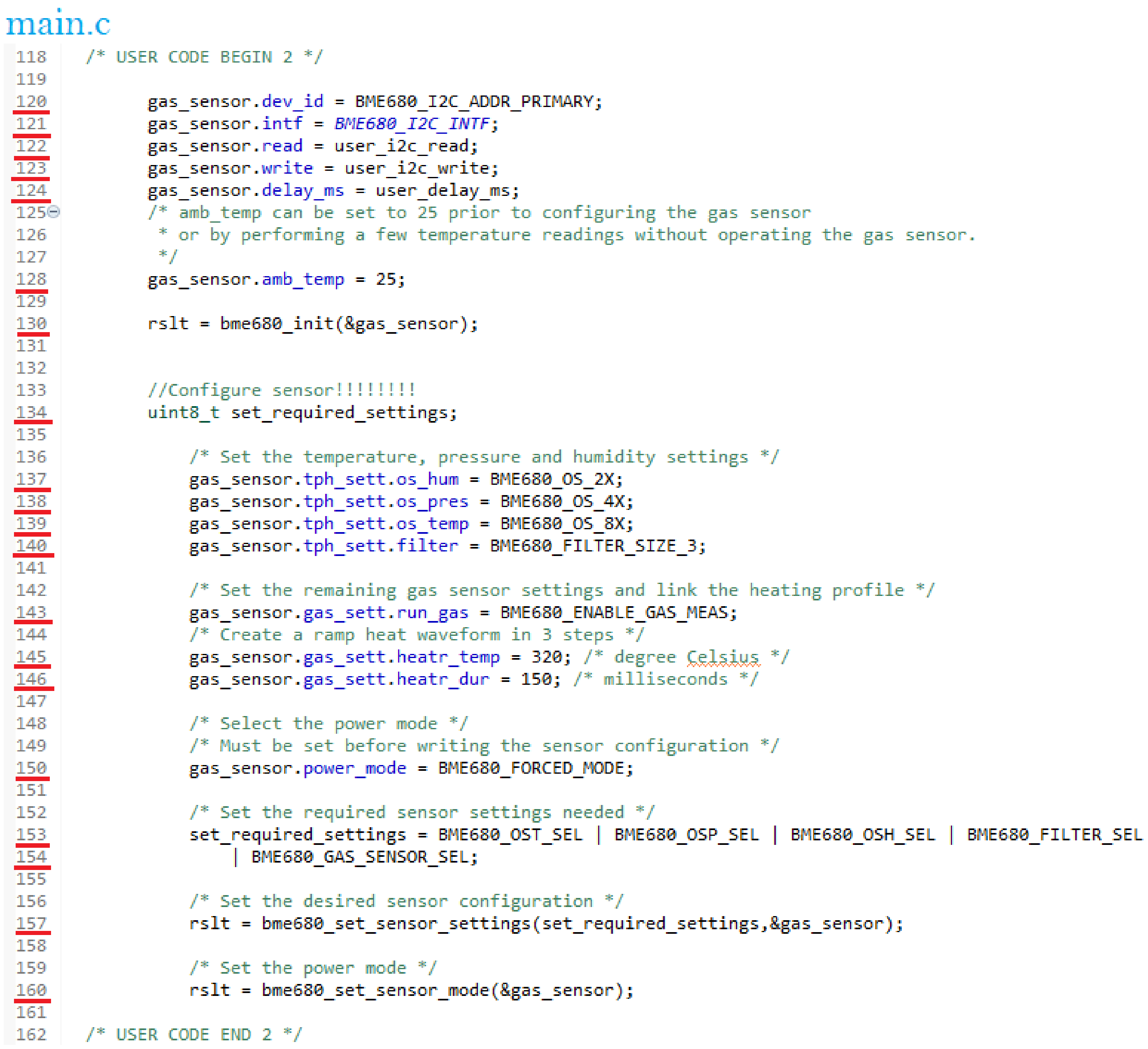
main.c

```
469
470  /* USER CODE BEGIN 4 */
471  void user_delay_ms(uint32_t period)     //Delay till the measurement is ready
472  {
473  HAL_Delay(period);
474  }
475
476
477  int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)     //I2C read
478  {
479      int8_t rslt = 0;
480      HAL_StatusTypeDef status = HAL_OK;
481      status = HAL_I2C_Mem_Read(&hi2c1, (uint16_t)(dev_id<<1), reg_addr, I2C_MEMADD_SIZE_8BIT, (uint8_t*)reg_data, len, 0x10000);
482      if(status != HAL_OK) rslt = -3;
483      return rslt;
484  }
485
486  int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)    //I2C write
487  {
488      int8_t rslt = 0;
489      HAL_StatusTypeDef status = HAL_OK;
490      status = HAL_I2C_Mem_Write(&hi2c1, (uint16_t)(dev_id<<1), reg_addr, I2C_MEMADD_SIZE_8BIT, (uint8_t*)reg_data, len, 0x10000);
491      if(status != HAL_OK) rslt = -3;
492      return rslt;
493  }
494  /* USER CODE END 4 */
495
497    * @brief  This function is executed in case of error occurrence.
```

Since we have defined the functions that we will use, we are ready to return to calibrate the sensor and ask for data outputs.

Before we ask for data from the sensor, we must activate and calibrate it, as it is described by the manufacturer.

Therefore, we add lines 120 till 160 as shown below. This part is given by the manufacturer in the **README.md** file that we have downloaded.


main.c

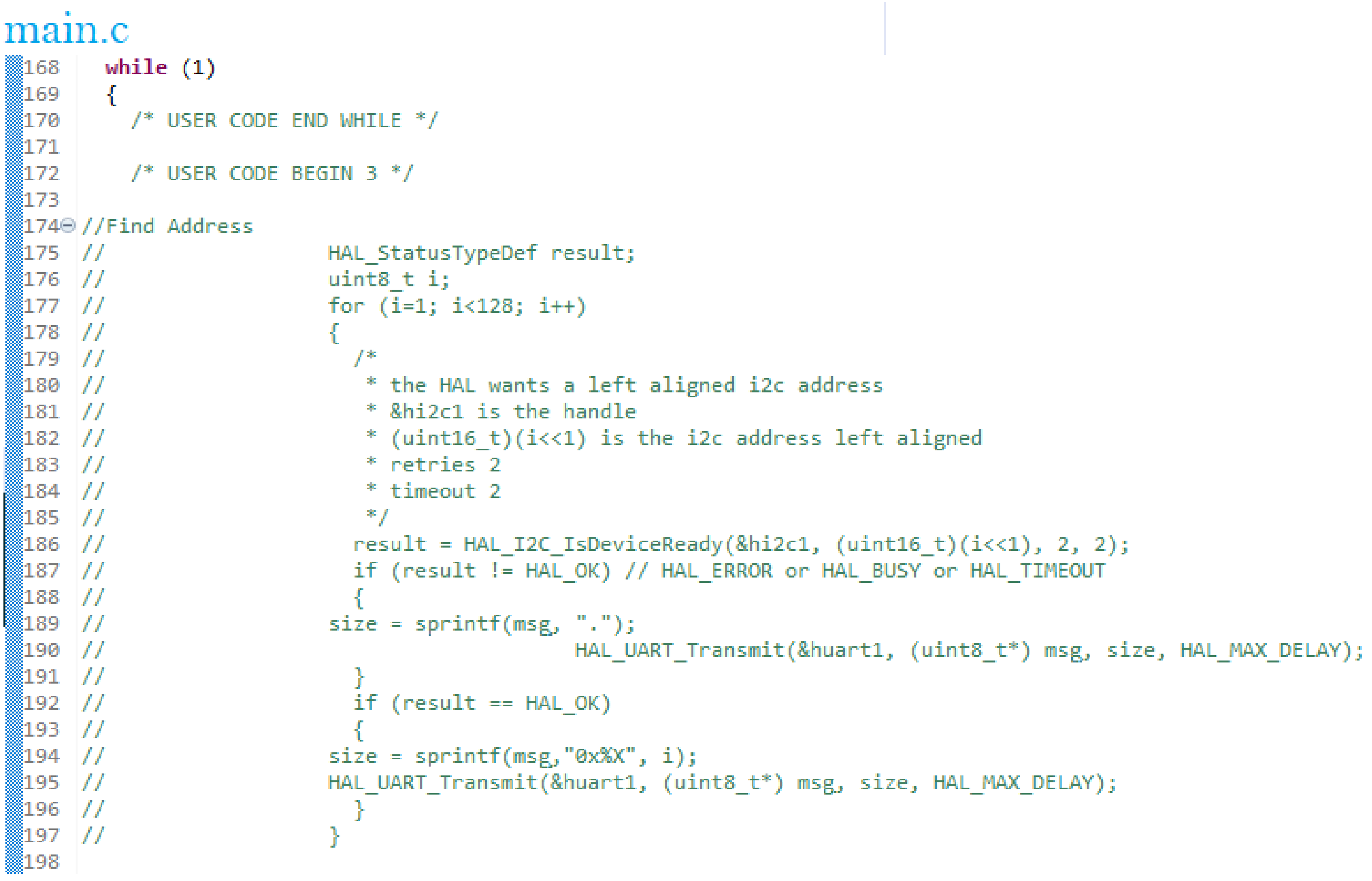
```
118  /* USER CODE BEGIN 2 */
119
120        gas_sensor.dev_id = BME680_I2C_ADDR_PRIMARY;
121        gas_sensor.intf = BME680_I2C_INTF;
122        gas_sensor.read = user_i2c_read;
123        gas_sensor.write = user_i2c_write;
124        gas_sensor.delay_ms = user_delay_ms;
125        /* amb_temp can be set to 25 prior to configuring the gas sensor
126         * or by performing a few temperature readings without operating the gas sensor.
127         */
128        gas_sensor.amb_temp = 25;
129
130        rslt = bme680_init(&gas_sensor);
131
132
133        //Configure sensor!!!!!!!!
134        uint8_t set_required_settings;
135
136            /* Set the temperature, pressure and humidity settings */
137            gas_sensor.tph_sett.os_hum = BME680_OS_2X;
138            gas_sensor.tph_sett.os_pres = BME680_OS_4X;
139            gas_sensor.tph_sett.os_temp = BME680_OS_8X;
140            gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;
141
142            /* Set the remaining gas sensor settings and link the heating profile */
143            gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
144            /* Create a ramp heat waveform in 3 steps */
145            gas_sensor.gas_sett.heatr_temp = 320; /* degree Celsius */
146            gas_sensor.gas_sett.heatr_dur = 150; /* milliseconds */
147
148            /* Select the power mode */
149            /* Must be set before writing the sensor configuration */
150            gas_sensor.power_mode = BME680_FORCED_MODE;
151
152            /* Set the required sensor settings needed */
153            set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL
154                | BME680_GAS_SENSOR_SEL;
155
156            /* Set the desired sensor configuration */
157            rslt = bme680_set_sensor_settings(set_required_settings,&gas_sensor);
158
159            /* Set the power mode */
160            rslt = bme680_set_sensor_mode(&gas_sensor);
161
162  /* USER CODE END 2 */
```

This part of the code it is activated first (once the device is activated) in order to activate and calibrate the sensor. It runs only once and then the program will enter the while(1) loop.

This next part inside the while(1) is not necessary for the project but we use it for debugging reasons.

Its purpose is to give as an uart1 output the address of each I2C devices that are connected with the STM32.

We can use to find an address or if we want to know if the microcontroller communicates with the sensor.



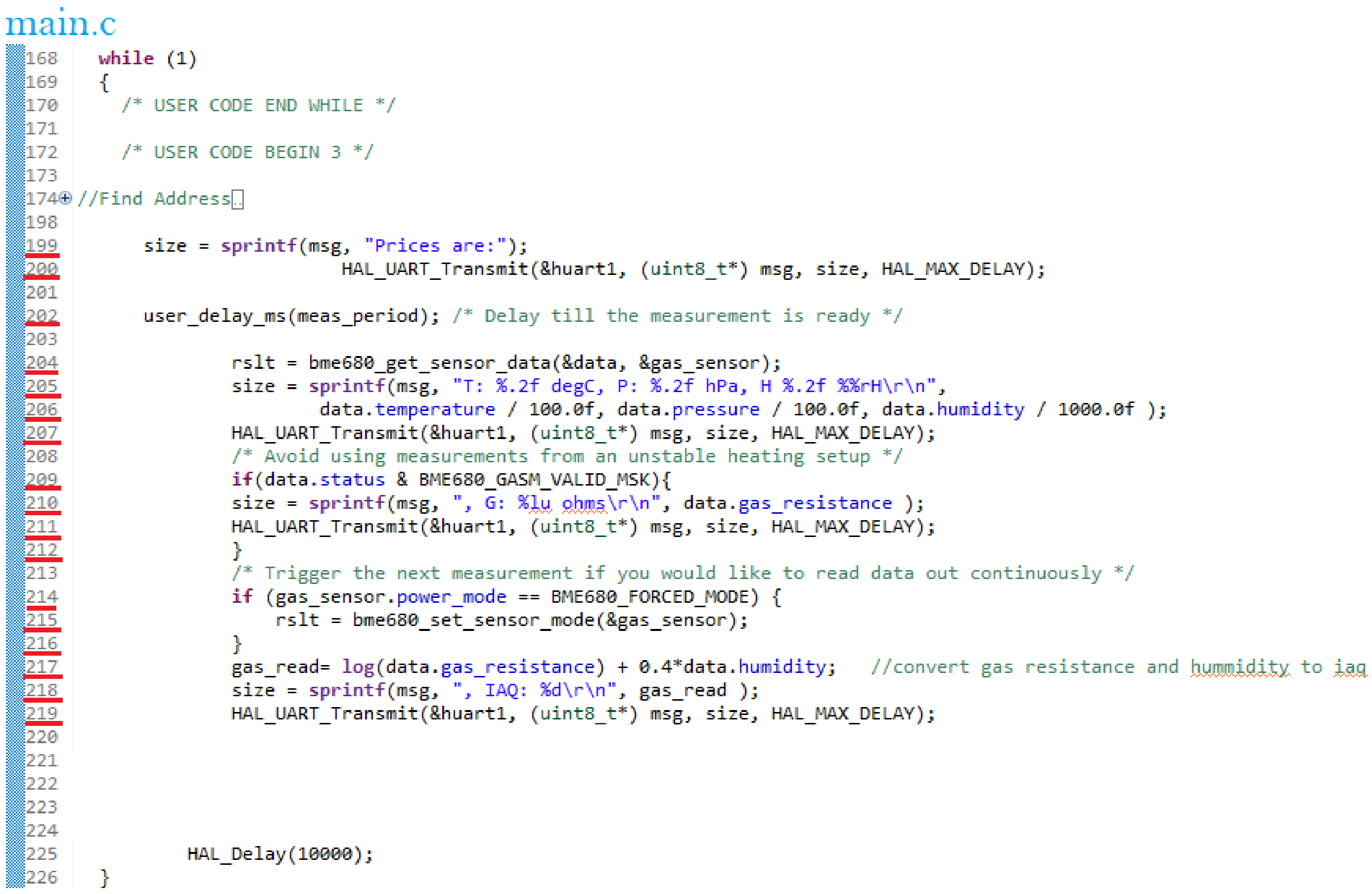
main.c

```c
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

//Find Address
//                  HAL_StatusTypeDef result;
//                  uint8_t i;
//                  for (i=1; i<128; i++)
//                  {
//                    /*
//                     * the HAL wants a left aligned i2c address
//                     * &hi2c1 is the handle
//                     * (uint16_t)(i<<1) is the i2c address left aligned
//                     * retries 2
//                     * timeout 2
//                     */
//                    result = HAL_I2C_IsDeviceReady(&hi2c1, (uint16_t)(i<<1), 2, 2);
//                    if (result != HAL_OK) // HAL_ERROR or HAL_BUSY or HAL_TIMEOUT
//                    {
//                  size = sprintf(msg, ".");
//                                    HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
//                    }
//                    if (result == HAL_OK)
//                    {
//                  size = sprintf(msg,"0x%X", i);
//                  HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
//                    }
//                  }

```

Its main function is to ping all available addresses and wait for a reply. If a device replies, it outputs the address through the uart1. If the address doesn’t return anything, it prints a “.”.As we mentioned above this part can be omitted.

Finally, we are ready to take measurements from the sensor. This next part will be inside the while(1) loop and continually will give measurements. There are parts that make use of the uart1, but those parts are only for debugging reasons. We can remove them. The same applies for the Delay at line 225.

So, we add lines 199 till 225.



main.c

```
168    while (1)
169    {
170      /* USER CODE END WHILE */
171
172      /* USER CODE BEGIN 3 */
173
174⊕ //Find Address
198
199        size = sprintf(msg, "Prices are:");
200                          HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
201
202        user_delay_ms(meas_period); /* Delay till the measurement is ready */
203
204                rslt = bme680_get_sensor_data(&data, &gas_sensor);
205                size = sprintf(msg, "T: %.2f degC, P: %.2f hPa, H %.2f %%rH\r\n",
206                        data.temperature / 100.0f, data.pressure / 100.0f, data.humidity / 1000.0f );
207                HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
208                /* Avoid using measurements from an unstable heating setup */
209                if(data.status & BME680_GASM_VALID_MSK){
210                size = sprintf(msg, ", G: %lu ohms\r\n", data.gas_resistance );
211                HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
212                }
213                /* Trigger the next measurement if you would like to read data out continuously */
214                if (gas_sensor.power_mode == BME680_FORCED_MODE) {
215                    rslt = bme680_set_sensor_mode(&gas_sensor);
216                }
217                gas_read= log(data.gas_resistance) + 0.4*data.humidity;   //convert gas resistance and hummidity to iaq
218                size = sprintf(msg, ", IAQ: %d\r\n", gas_read );
219                HAL_UART_Transmit(&huart1, (uint8_t*) msg, size, HAL_MAX_DELAY);
220
221
222
223
224
225            HAL_Delay(10000);
226    }
```

The measurements that we receive from the bme680 sensor are:

- data.temperature =>Temperature(C°)
- data.pressure =>Pressure (hPa)
- data.humidity =>Humidity (%)
- data.gas_resistance =>Gas Resistance (Ohm)

But the measurement that we want is the **IAQ** (Index of Air Quality).

To calculate it, we use function:

$$\boldsymbol{IAQ = \log(gas\ resistance) + 0.4humidity}$$

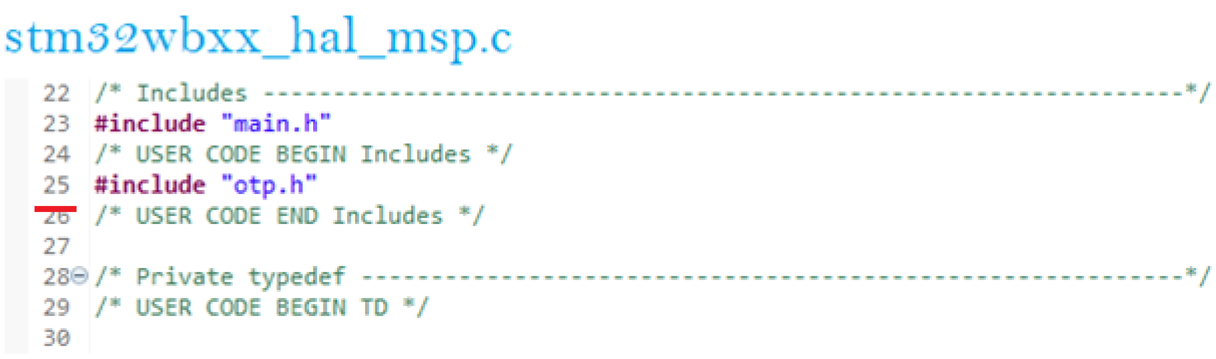
5.3 BLE Code

Since we have the bme680 code up and working, it is time to set the BLE function in our project.

So, the first thing that we must do is, to tune our HSP (High Speed Clock). We start by setting the otp (One-Time-Programmable) code. They can be used for permanent store of configuration data for your device.

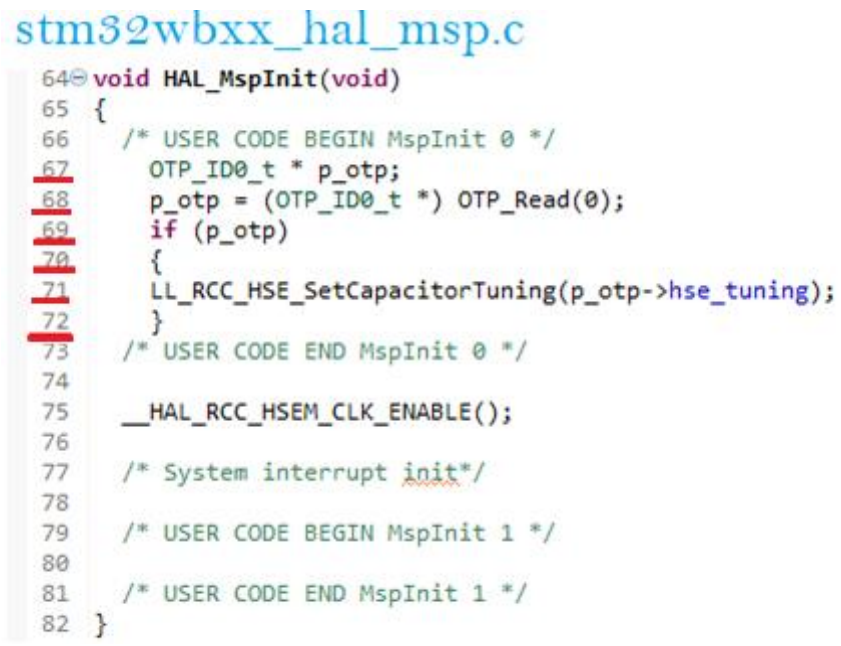
The manufacturer provides this info and we simply add it to the code.

We add line 25 and lines 67 till 72 in stm32wbxx_hal_msp.c class.



stm32wbxx_hal_msp.c

```
22 /* Includes ------------------------------------------------------------------*/
23 #include "main.h"
24 /* USER CODE BEGIN Includes */
25 #include "otp.h"
26 /* USER CODE END Includes */
27
28 /* Private typedef -----------------------------------------------------------*/
29 /* USER CODE BEGIN TD */
30
```



stm32wbxx_hal_msp.c

```
64 void HAL_MspInit(void)
65 {
66   /* USER CODE BEGIN MspInit 0 */
67     OTP_ID0_t * p_otp;
68     p_otp = (OTP_ID0_t *) OTP_Read(0);
69     if (p_otp)
70     {
71     LL_RCC_HSE_SetCapacitorTuning(p_otp->hse_tuning);
72     }
73   /* USER CODE END MspInit 0 */
74
75   __HAL_RCC_HSEM_CLK_ENABLE();
76
77   /* System interrupt init*/
78
79   /* USER CODE BEGIN MspInit 1 */
80
81   /* USER CODE END MspInit 1 */
82 }
```

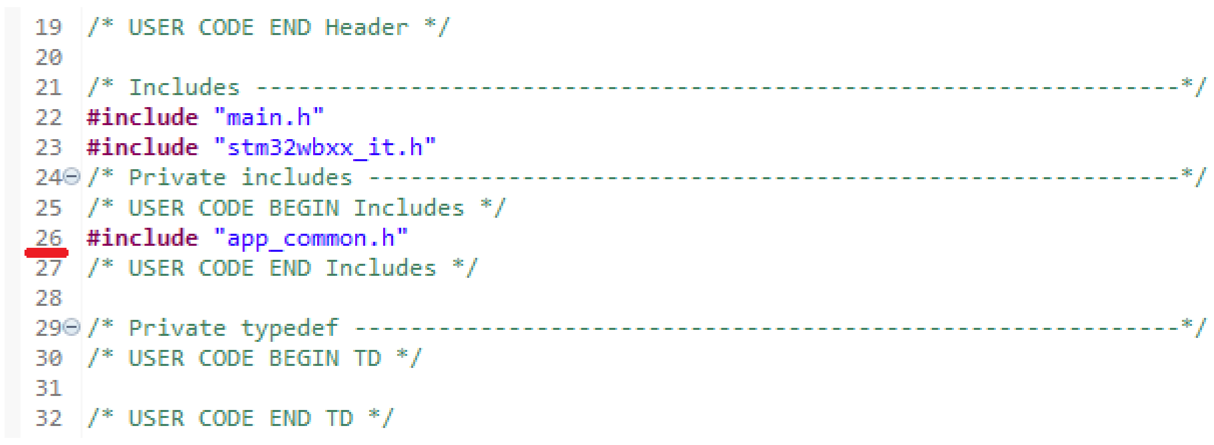
Next, we need to put the interrupt service routine in. So, the IPPC is doing the interrupts so the Cortex m4 can communicate with the Cortex m0, and we also need to put in the wakeup handler for Our RTC.

We do this so our software interrupts and time servers from the library stack can be used inside the application.

We make this addition inside the stm32wbxx_it.c

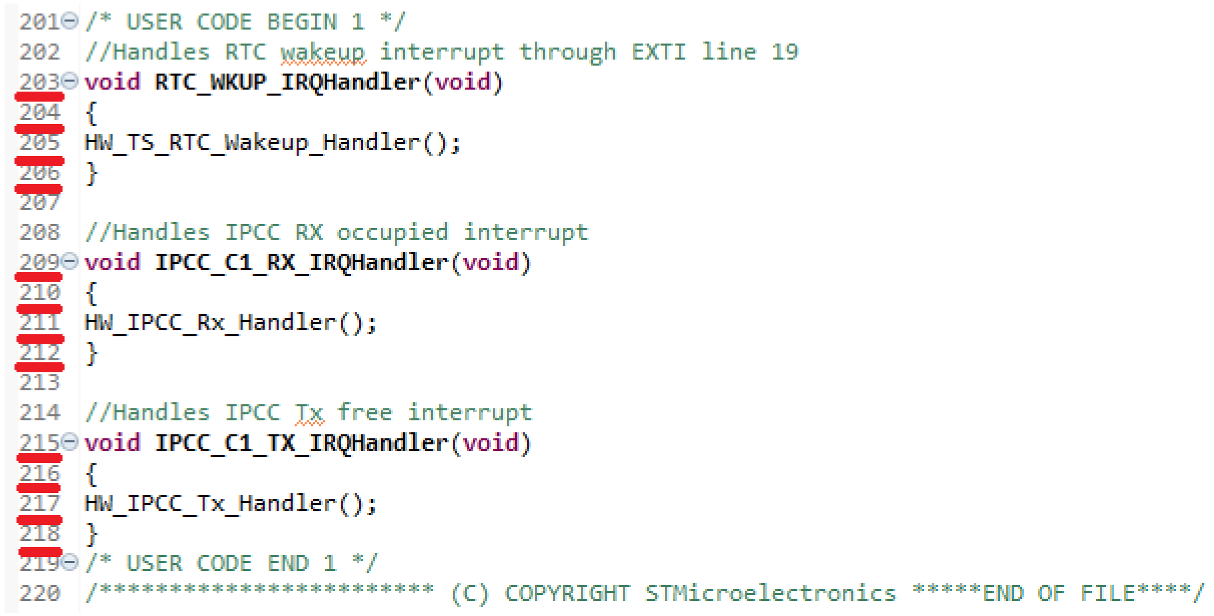
We add lines 26 and 203 till 218 inside the stm32wbxx_it.c class.

stm32wbxx_it.c



```
19  /* USER CODE END Header */
20
21  /* Includes ------------------------------------------------------------------*/
22  #include "main.h"
23  #include "stm32wbxx_it.h"
24⊖ /* Private includes ----------------------------------------------------------*/
25  /* USER CODE BEGIN Includes */
26  #include "app_common.h"
27  /* USER CODE END Includes */
28
29⊖ /* Private typedef -----------------------------------------------------------*/
30  /* USER CODE BEGIN TD */
31
32  /* USER CODE END TD */
```

stm32wbxx_it.c

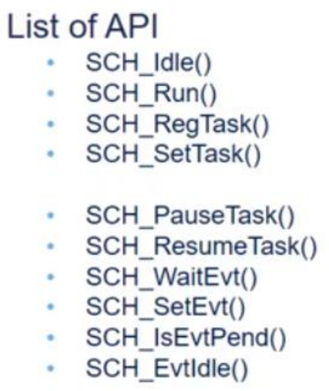


```
201⊖ /* USER CODE BEGIN 1 */
202  //Handles RTC wakeup interrupt through EXTI line 19
203⊖ void RTC_WKUP_IRQHandler(void)
204  {
205  HW_TS_RTC_Wakeup_Handler();
206  }
207
208  //Handles IPCC RX occupied interrupt
209⊖ void IPCC_C1_RX_IRQHandler(void)
210  {
211  HW_IPCC_Rx_Handler();
212  }
213
214  //Handles IPCC Tx free interrupt
215⊖ void IPCC_C1_TX_IRQHandler(void)
216  {
217  HW_IPCC_Tx_Handler();
218  }
219⊖ /* USER CODE END 1 */
220  /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

The typical architecture that we have is based on a Sequencer (or Simple Task Scheduler). That mean that we set various tasks and what goes through we have a switch statement (for all our tasks). Eventually we get to the point that all the tasks are complete therefore the device goes to case idle which means enter low power mode.

Inside the Scheduler (inside the utilities folder) we can have up to 32 different tasks, the ability to request a task to be executed or pause and resume, or even wait for specific events to happen to be executed.

Here is a list of API commands for various tasks:

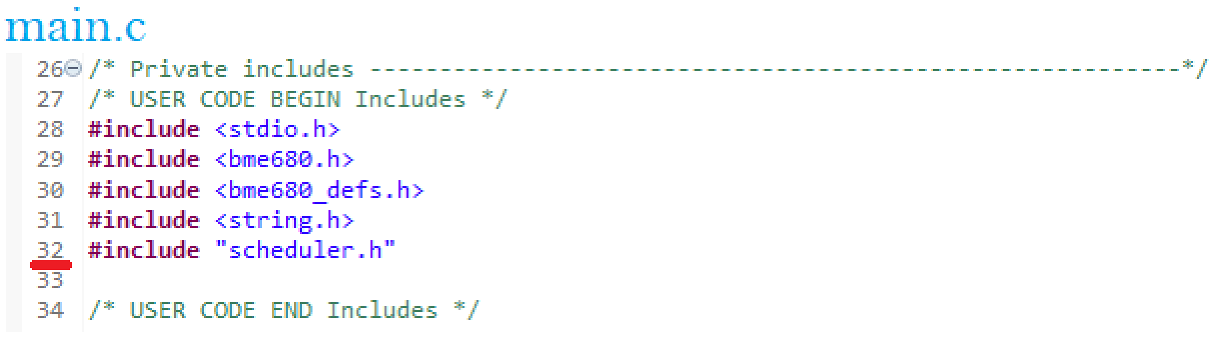


List of API

- SCH_Idle()
- SCH_Run()
- SCH_RegTask()
- SCH_SetTask()

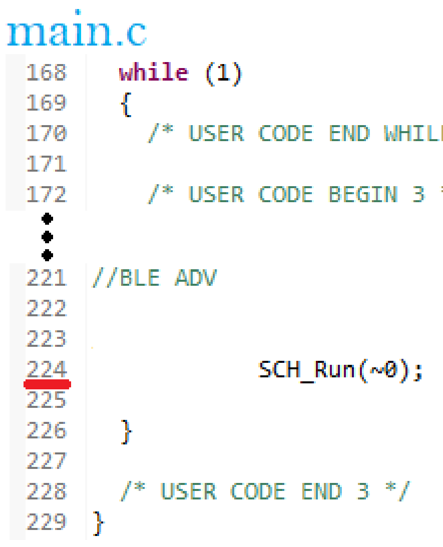
- SCH_PauseTask()
- SCH_ResumeTask()
- SCH_WaitEvt()
- SCH_SetEvt()
- SCH_IsEvtPend()
- SCH_EvtIdle()

To include the Scheduler to our main.c we add the lines 32 and 224.



main.c

```
26 /* Private includes ----------------------------------------------------------*/
27 /* USER CODE BEGIN Includes */
28 #include <stdio.h>
29 #include <bme680.h>
30 #include <bme680_defs.h>
31 #include <string.h>
32 #include "scheduler.h"
33
34 /* USER CODE END Includes */
```

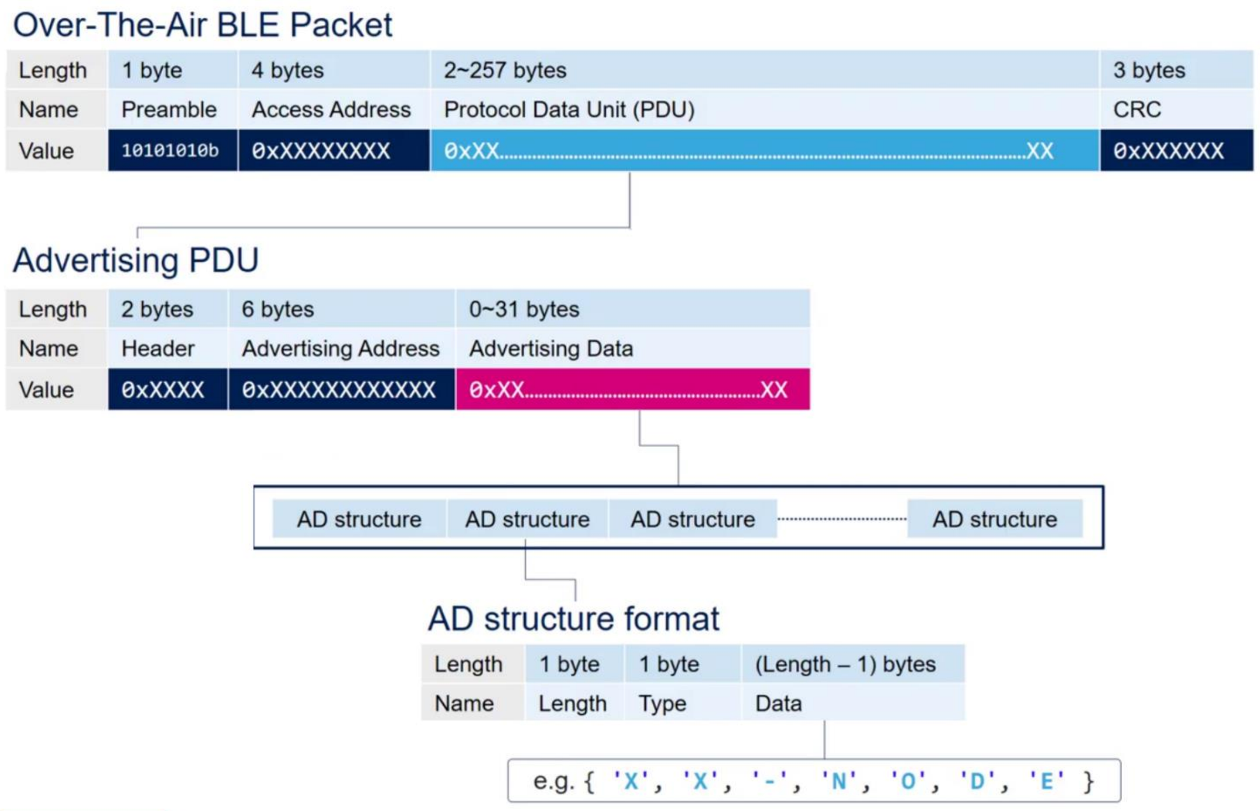


main.c

```
168    while (1)
169    {
170      /* USER CODE END WHILE */
171
172      /* USER CODE BEGIN 3 */
  ⋮
221  //BLE ADV
222
223
224              SCH_Run(~0);
225
226    }
227
228    /* USER CODE END 3 */
229  }
```

The BLE Advertising data is a defined string and has certain parameters according to the BLE protocols. We can configure the advertising data which is part of the Protocol data unit.

Inside the Advertising data there is a predefined structure, and it tells us certain things about manufacturer specific data and where our local name is.



Over-The-Air BLE Packet

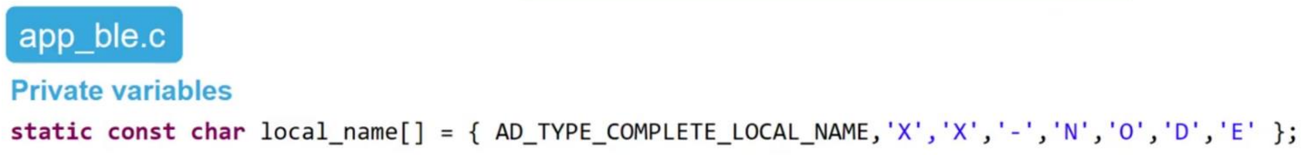
Length	1 byte	4 bytes	2~257 bytes	3 bytes
Name	Preamble	Access Address	Protocol Data Unit (PDU)	CRC
Value	10101010b	0xXXXXXXXX	0xXX.......XX	0xXXXXXX

Advertising PDU

Length	2 bytes	6 bytes	0~31 bytes
Name	Header	Advertising Address	Advertising Data
Value	0xXXXX	0xXXXXXXXXXXXX	0xXX.......XX

AD structure format

Length	1 byte	1 byte	(Length – 1) bytes
Name	Length	Type	Data

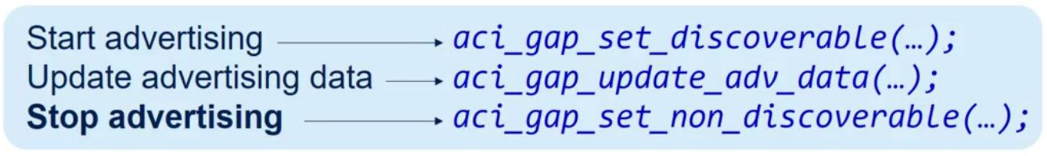


app_ble.c

Private variables

```
static const char local_name[] = { AD_TYPE_COMPLETE_LOCAL_NAME,'X','X','-','N','O','D','E' };
```

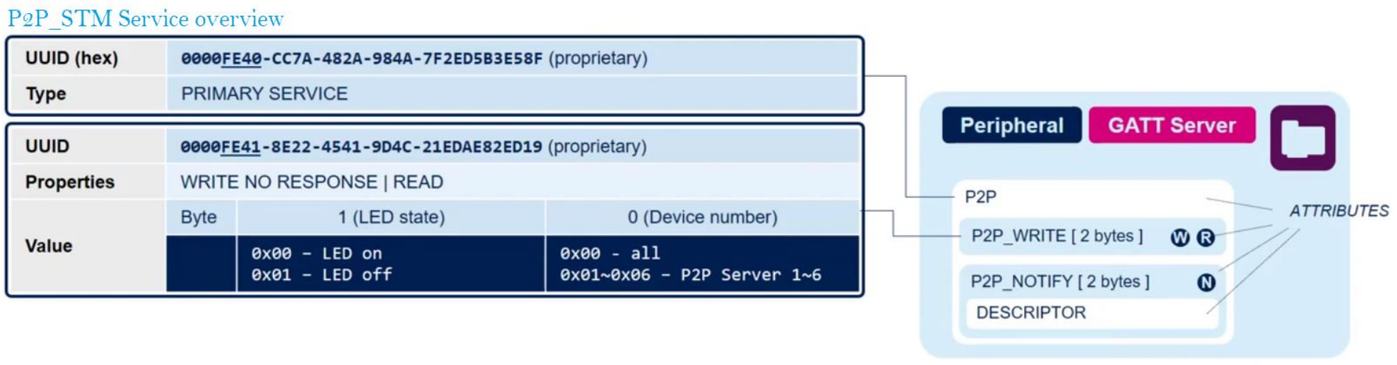
Inside the class app_ble.c are all the necessary code for the correct function of the Bluetooth. It already contains the following commands:



Start advertising →	`aci_gap_set_discoverable(…);`
Update advertising data →	`aci_gap_update_adv_data(…);`
Stop advertising →	`aci_gap_set_non_discoverable(…);`

As we mentioned above, we will add a led for debugging reasons. To do so we must offer a new service.

This will be a P2P service, a primary service.

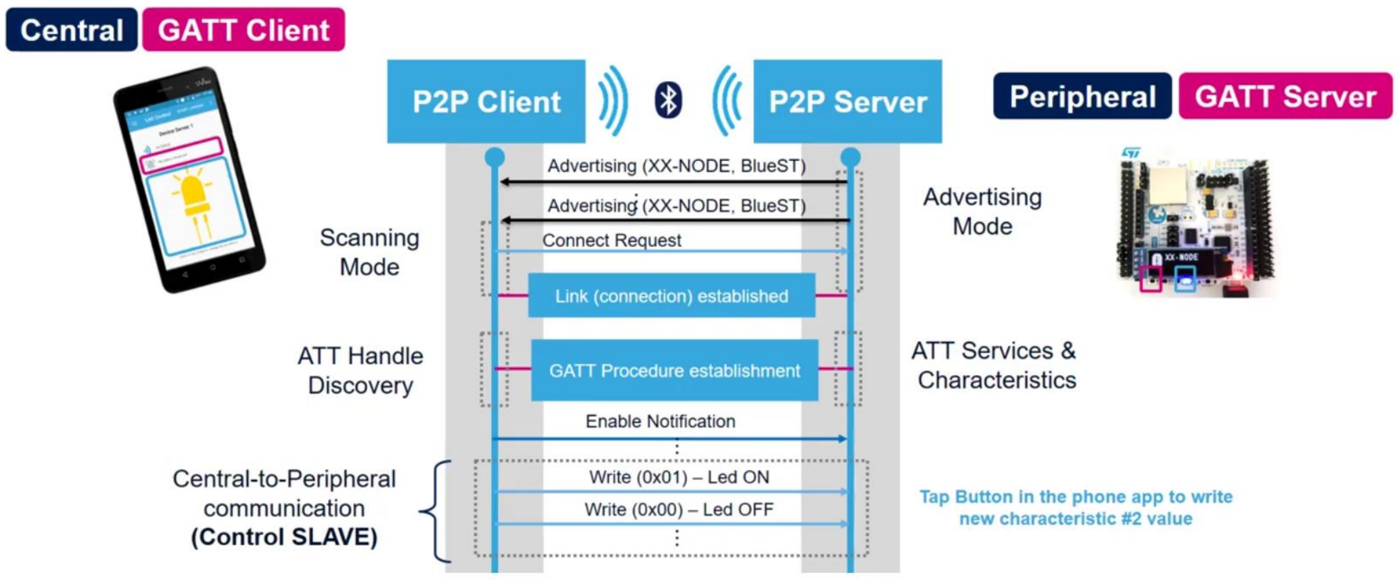


P2P_STM Service overview

UUID (hex)	0000FE40-CC7A-482A-984A-7F2ED5B3E58F (proprietary)	
Type	PRIMARY SERVICE	

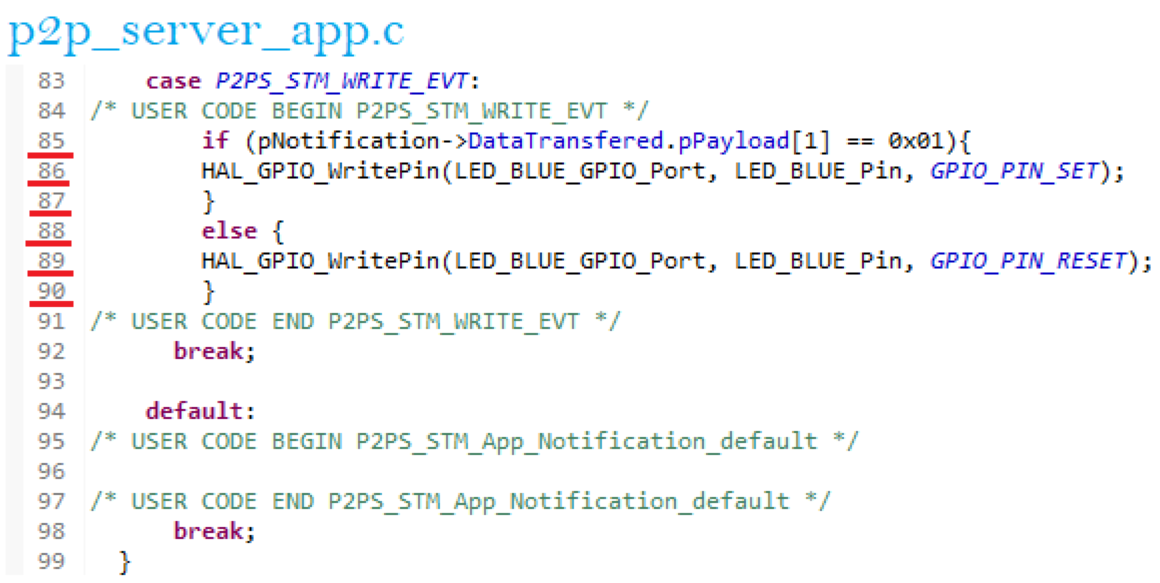
UUID	0000FE41-8E22-4541-9D4C-21EDAE82ED19 (proprietary)		
Properties	WRITE NO RESPONSE | READ		
Value	Byte	1 (LED state)	0 (Device number)
		0x00 – LED on 0x01 – LED off	0x00 - all 0x01~0x06 – P2P Server 1~6

By this point this is the communication between Client and Server. The Led will be controlled through a phone application that we will cover in another Chapter.


Central
GATT Client
P2P Client
P2P Server
Peripheral
GATT Server
Scanning Mode
Advertising (XX-NODE, BlueST)
Advertising (XX-NODE, BlueST)
Connect Request
Advertising Mode
Link (connection) established
ATT Handle Discovery
GATT Procedure establishment
ATT Services & Characteristics
Enable Notification
Central-to-Peripheral communication (Control SLAVE)
Write (0x01) – Led ON
Write (0x00) – Led OFF
Tap Button in the phone app to write new characteristic #2 value

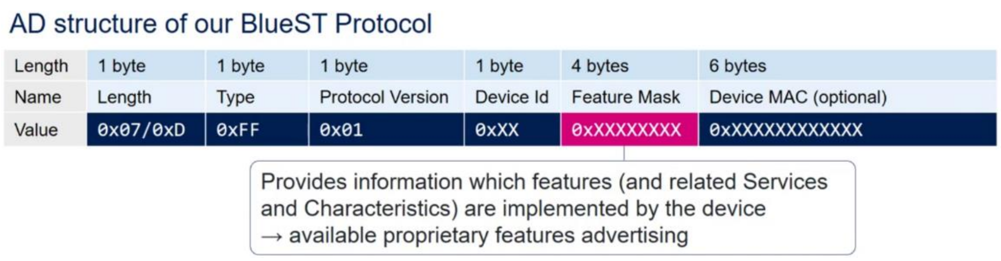
To set the code for the Led, we must input the following code into p2p_server_app.c

We add lines 85 till 90 into p2p_server_app.c


p2p_server_app.c

```
83      case P2PS_STM_WRITE_EVT:
84  /* USER CODE BEGIN P2PS_STM_WRITE_EVT */
85          if (pNotification->DataTransfered.pPayload[1] == 0x01){
86          HAL_GPIO_WritePin(LED_BLUE_GPIO_Port, LED_BLUE_Pin, GPIO_PIN_SET);
87          }
88          else {
89          HAL_GPIO_WritePin(LED_BLUE_GPIO_Port, LED_BLUE_Pin, GPIO_PIN_RESET);
90          }
91  /* USER CODE END P2PS_STM_WRITE_EVT */
92        break;
93
94      default:
95  /* USER CODE BEGIN P2PS_STM_App_Notification_default */
96
97  /* USER CODE END P2PS_STM_App_Notification_default */
98        break;
99    }
```

Now we must create a new service for our bme680. But the Client must receive information about what it is transmitted in our Advertising Data in the form of a mask.

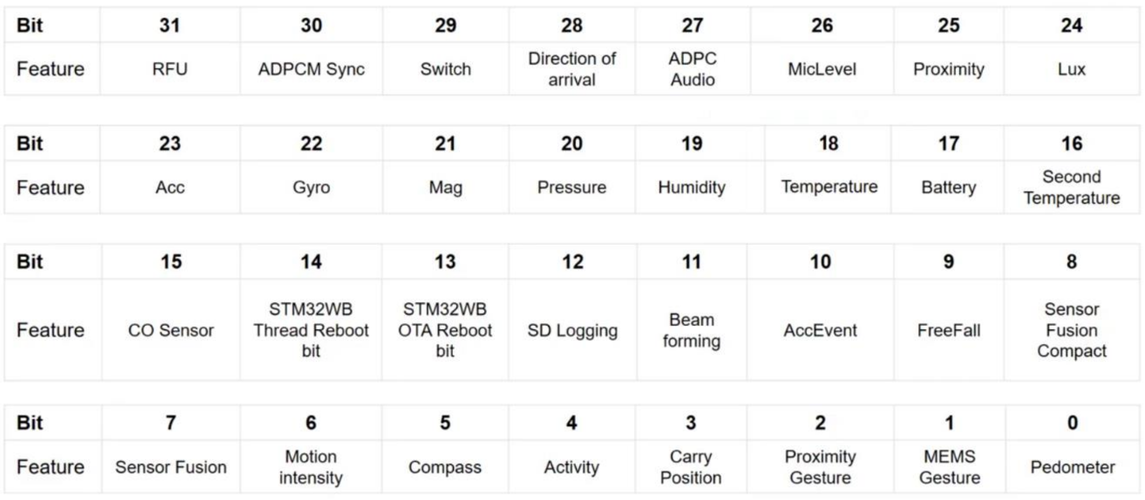


AD structure of our BlueST Protocol

Length	1 byte	1 byte	1 byte	1 byte	4 bytes	6 bytes
Name	Length	Type	Protocol Version	Device Id	Feature Mask	Device MAC (optional)
Value	0x07/0xD	0xFF	0x01	0xXX	0xXXXXXXXX	0xXXXXXXXXXXXX

Provides information which features (and related Services and Characteristics) are implemented by the device
→ available proprietary features advertising

We set this mask accordingly with the feature masks that are available from the data reading app developer.



Bit	31	30	29	28	27	26	25	24
Feature	RFU	ADPCM Sync	Switch	Direction of arrival	ADPC Audio	MicLevel	Proximity	Lux

Bit	23	22	21	20	19	18	17	16
Feature	Acc	Gyro	Mag	Pressure	Humidity	Temperature	Battery	Second Temperature

Bit	15	14	13	12	11	10	9	8
Feature	CO Sensor	STM32WB Thread Reboot bit	STM32WB OTA Reboot bit	SD Logging	Beam forming	AccEvent	FreeFall	Sensor Fusion Compact

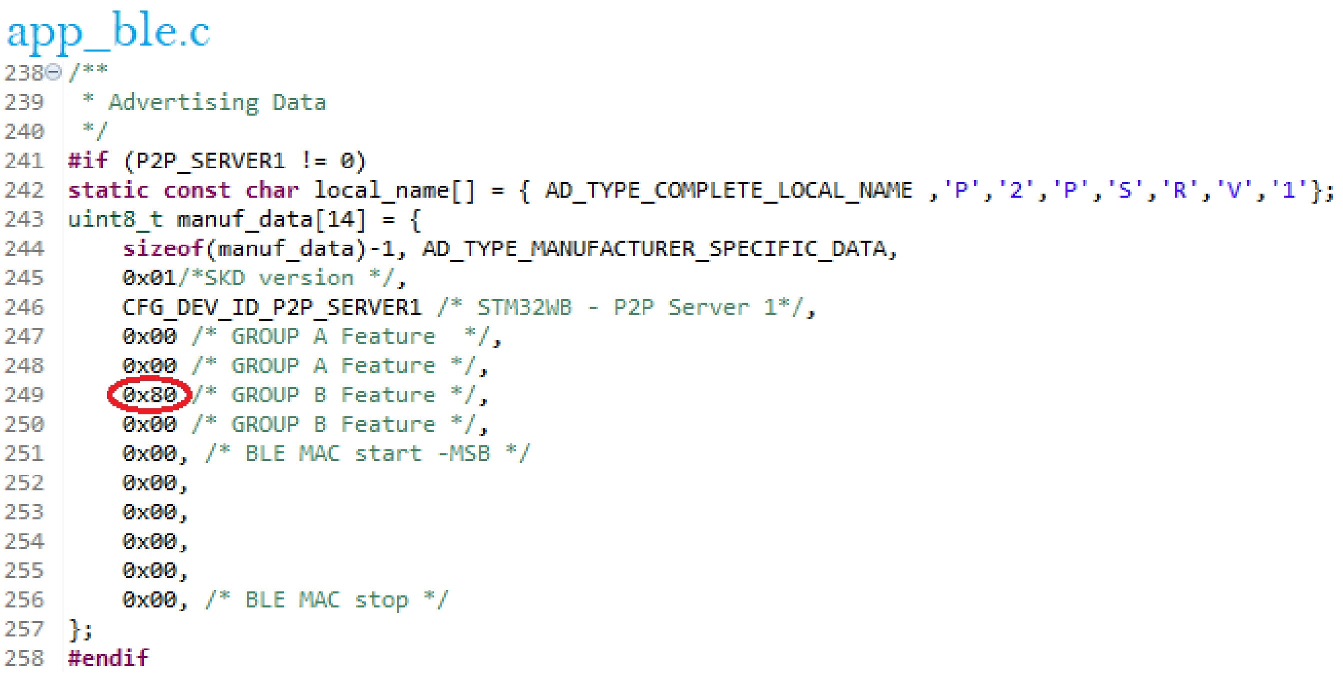
Bit	7	6	5	4	3	2	1	0
Feature	Sensor Fusion	Motion intensity	Compass	Activity	Carry Position	Proximity Gesture	MEMS Gesture	Pedometer

Since we want to send CO, we will use bit 15.

We must configure the mask in the Advertising part of app_ble.c. Here we have 4 bytes which means 32 bit. That means, that in order to have a CO mask, the bit 15 must change to 1.

00000000 00000000 10000000 00000000 = 00 00 80 00 in Hex

We change the line 249 from 0x00 to 0x80

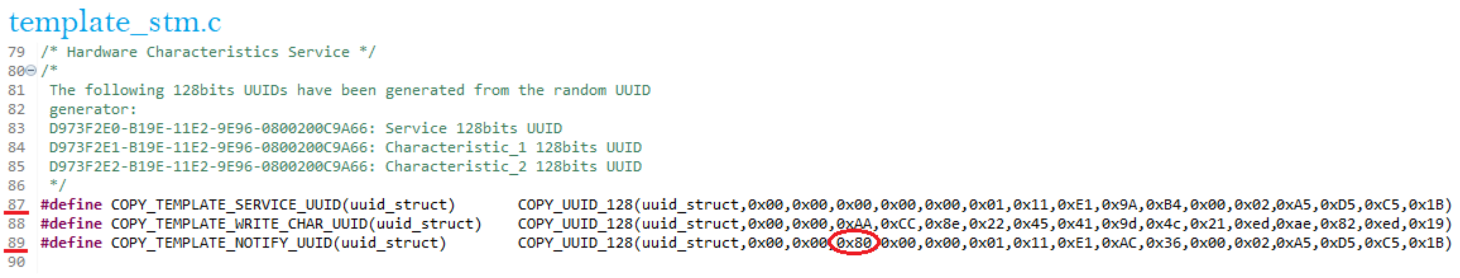


app_ble.c

```
238 /**
239   * Advertising Data
240   */
241 #if (P2P_SERVER1 != 0)
242 static const char local_name[] = { AD_TYPE_COMPLETE_LOCAL_NAME ,'P','2','P','S','R','V','1'};
243 uint8_t manuf_data[14] = {
244     sizeof(manuf_data)-1, AD_TYPE_MANUFACTURER_SPECIFIC_DATA,
245     0x01/*SKD version */,
246     CFG_DEV_ID_P2P_SERVER1 /* STM32WB - P2P Server 1*/,
247     0x00 /* GROUP A Feature  */,
248     0x00 /* GROUP A Feature */,
249     0x80 /* GROUP B Feature */,
250     0x00 /* GROUP B Feature */,
251     0x00, /* BLE MAC start -MSB */
252     0x00,
253     0x00,
254     0x00,
255     0x00,
256     0x00, /* BLE MAC stop */
257 };
258 #endif
```

We have set the mask, but it is still classed as an unknown service. That is why we must make the service recognizable for the client. We will modify the service UUID in template_stm.c.

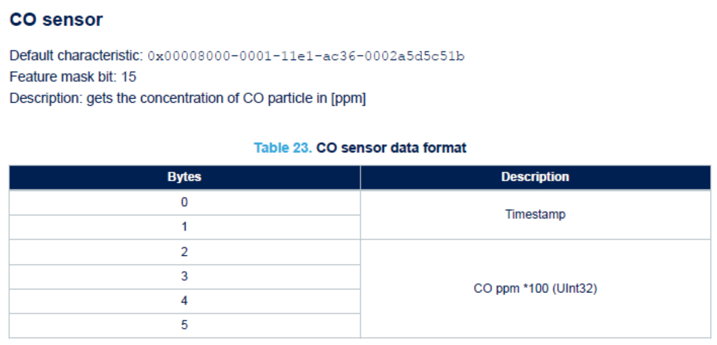
At line 87 we give a unique UUID and at line 89 a unique characteristic UUID. At 89 we also add the mask for CO in Hex.



template_stm.c

```
79  /* Hardware Characteristics Service */
80⊖ /*
81   The following 128bits UUIDs have been generated from the random UUID
82   generator:
83   D973F2E0-B19E-11E2-9E96-0800200C9A66: Service 128bits UUID
84   D973F2E1-B19E-11E2-9E96-0800200C9A66: Characteristic_1 128bits UUID
85   D973F2E2-B19E-11E2-9E96-0800200C9A66: Characteristic_2 128bits UUID
86   */
87  #define COPY_TEMPLATE_SERVICE_UUID(uuid_struct)      COPY_UUID_128(uuid_struct,0x00,0x00,0x00,0x00,0x00,0x01,0x11,0xE1,0x9A,0xB4,0x00,0x02,0xA5,0xD5,0xC5,0x1B)
88  #define COPY_TEMPLATE_WRITE_CHAR_UUID(uuid_struct)   COPY_UUID_128(uuid_struct,0x00,0x00,0xAA,0xCC,0x8e,0x22,0x45,0x41,0x9d,0x4c,0x21,0xed,0xae,0x82,0xed,0x19)
89  #define COPY_TEMPLATE_NOTIFY_UUID(uuid_struct)       COPY_UUID_128(uuid_struct,0x00,0x00,0x80,0x00,0x00,0x01,0x11,0xE1,0xAC,0x36,0x00,0x02,0xA5,0xD5,0xC5,0x1B)
90
```

The Packet that we want to send is 6 bytes, 2 bytes as a timestamp and 4 bytes for the CO reading, according with the BlueST protocol.



CO sensor

Default characteristic: 0x00008000-0001-11e1-ac36-0002a5d5c51b

Feature mask bit: 15

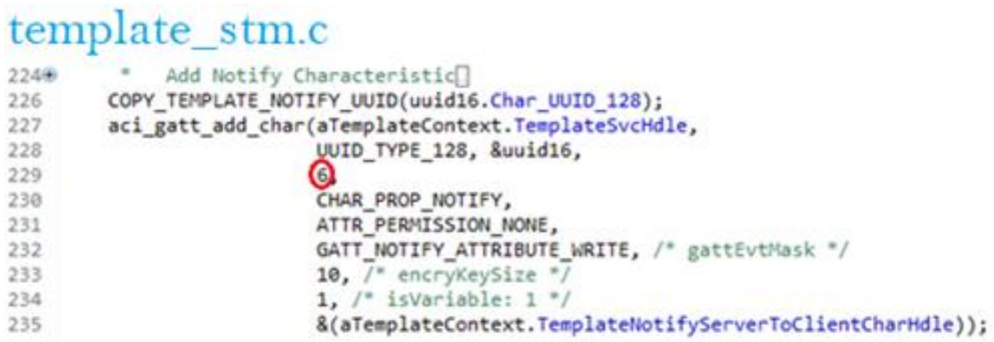
Description: gets the concentration of CO particle in [ppm]

Table 23. **CO sensor data format**

Bytes	Description
0	Timestamp
1	
2	CO ppm *100 (UInt32)
3	
4	
5	

This means that we must align the Service Characteristics with the GATT client expectations. We will make this change at the template_stm.c

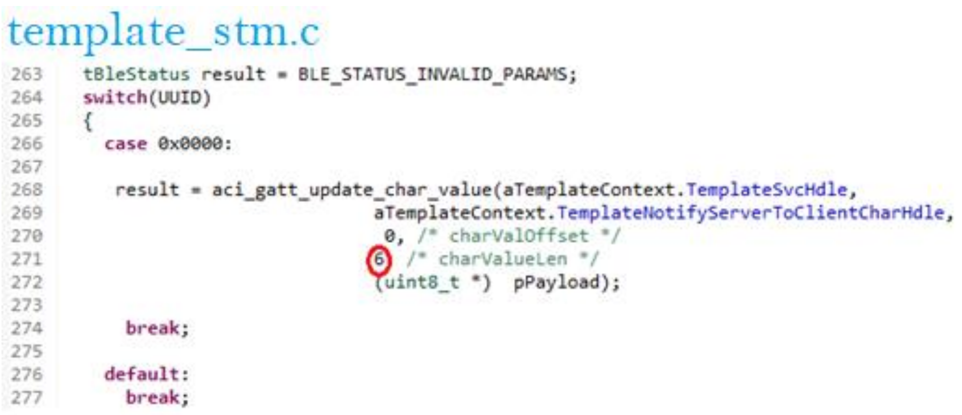
At line 229 we change 2 to 6 bytes.



template_stm.c

```
224⊕     *  Add Notify Characteristic[]
226      COPY_TEMPLATE_NOTIFY_UUID(uuid16.Char_UUID_128);
227      aci_gatt_add_char(aTemplateContext.TemplateSvcHdle,
228                        UUID_TYPE_128, &uuid16,
229                        6,
230                        CHAR_PROP_NOTIFY,
231                        ATTR_PERMISSION_NONE,
232                        GATT_NOTIFY_ATTRIBUTE_WRITE, /* gattEvtMask */
233                        10, /* encryKeySize */
234                        1, /* isVariable: 1 */
235                        &(aTemplateContext.TemplateNotifyServerToClientCharHdle));
```

Also make the same change at line 271.

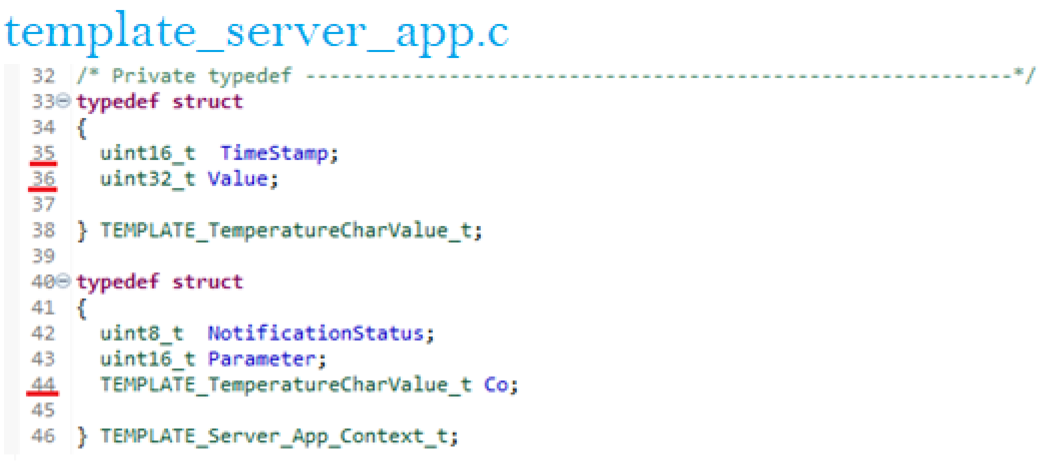


template_stm.c

```
263  tBleStatus result = BLE_STATUS_INVALID_PARAMS;
264  switch(UUID)
265  {
266    case 0x0000:
267
268     result = aci_gatt_update_char_value(aTemplateContext.TemplateSvcHdle,
269                              aTemplateContext.TemplateNotifyServerToClientCharHdle,
270                               0, /* charValOffset */
271                               6  /* charValueLen */
272                              (uint8_t *)  pPayload);
273
274      break;
275
276    default:
277      break;
```

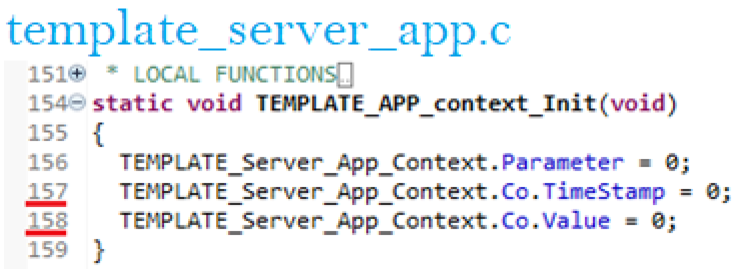
We have finished the services part we are moving to the application part. Now we can begin to build the structure for these 6 bytes. We move to the file named template_server_app.c

We add lines 35,36 and 44.


template_server_app.c

```
32  /* Private typedef -----------------------------------------------------------*/
33  typedef struct
34  {
35    uint16_t  TimeStamp;
36    uint32_t Value;
37
38  } TEMPLATE_TemperatureCharValue_t;
39
40  typedef struct
41  {
42    uint8_t  NotificationStatus;
43    uint16_t Parameter;
44    TEMPLATE_TemperatureCharValue_t Co;
45
46  } TEMPLATE_Server_App_Context_t;
```

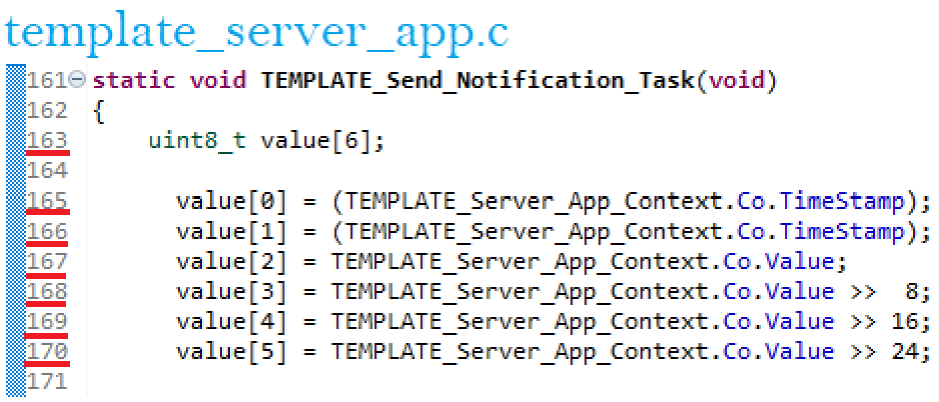
Next, we will Initialize the new app context variables. We add lines 157 and 158.


template_server_app.c

```
151   * LOCAL FUNCTIONS
154 static void TEMPLATE_APP_context_Init(void)
155 {
156   TEMPLATE_Server_App_Context.Parameter = 0;
157   TEMPLATE_Server_App_Context.Co.TimeStamp = 0;
158   TEMPLATE_Server_App_Context.Co.Value = 0;
159 }
```

Now we need to send the update characteristic function. When we send our notify, there is a task happening to send all our bytes.

We add lines 163 and 165 till 170.

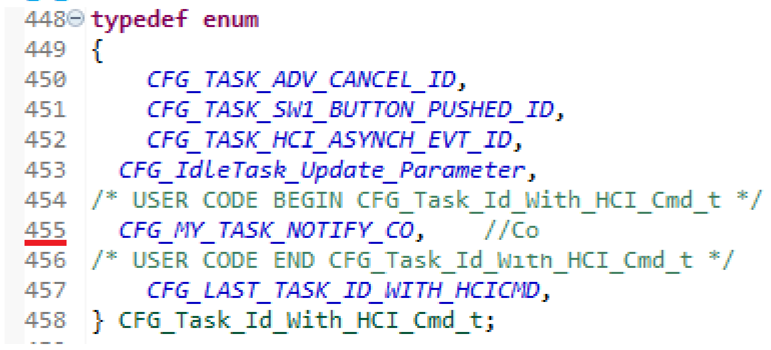

template_server_app.c

```
161 static void TEMPLATE_Send_Notification_Task(void)
162 {
163     uint8_t value[6];
164
165       value[0] = (TEMPLATE_Server_App_Context.Co.TimeStamp);
166       value[1] = (TEMPLATE_Server_App_Context.Co.TimeStamp);
167       value[2] = TEMPLATE_Server_App_Context.Co.Value;
168       value[3] = TEMPLATE_Server_App_Context.Co.Value >>  8;
169       value[4] = TEMPLATE_Server_App_Context.Co.Value >> 16;
170       value[5] = TEMPLATE_Server_App_Context.Co.Value >> 24;
171
```

Moving on, we need to register a task to be used to update characteristics. We register a notify CO2 task. We create a task Id at app_conf.h and create a task initialization at template_server_app.c

We add line 455.

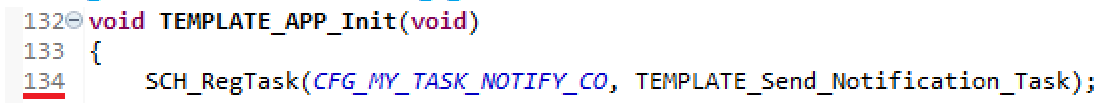
app_conf.h



```
448⊖typedef enum
449 {
450       CFG_TASK_ADV_CANCEL_ID,
451       CFG_TASK_SW1_BUTTON_PUSHED_ID,
452       CFG_TASK_HCI_ASYNCH_EVT_ID,
453     CFG_IdleTask_Update_Parameter,
454 /* USER CODE BEGIN CFG_Task_Id_With_HCI_Cmd_t */
455     CFG_MY_TASK_NOTIFY_CO,     //Co
456 /* USER CODE END CFG_Task_Id_With_HCI_Cmd_t */
457       CFG_LAST_TASK_ID_WITH_HCICMD,
458 } CFG_Task_Id_With_HCI_Cmd_t;
```

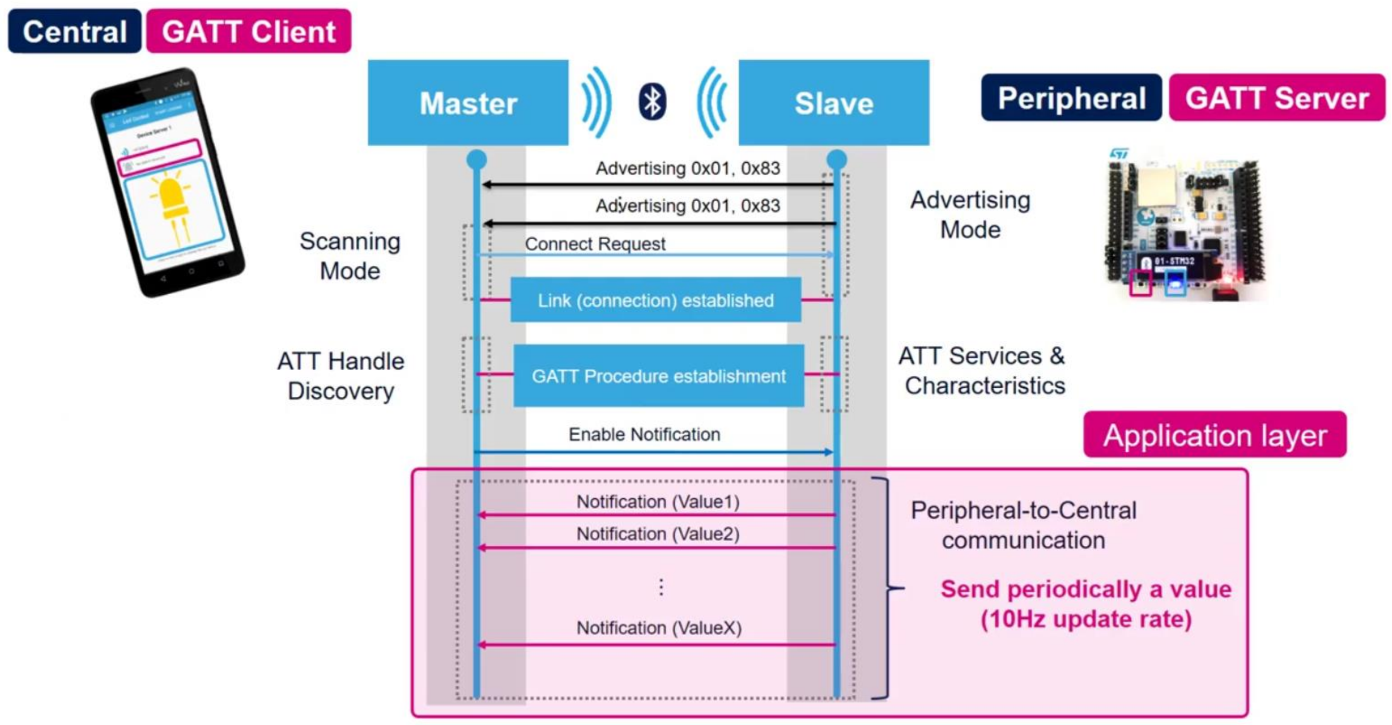
We add line 134.

template_server_app.c

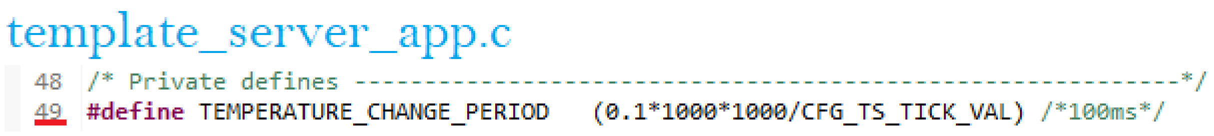


```
132⊖void TEMPLATE_APP_Init(void)
133 {
134     SCH_RegTask(CFG_MY_TASK_NOTIFY_CO, TEMPLATE_Send_Notification_Task);
```

The image below shows the current function of the system. The next step that we will perform will set a 100ms refresh rate.

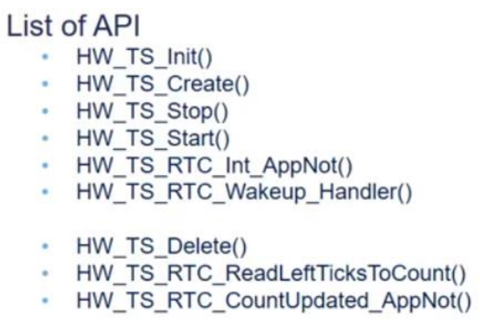


We add line 49 at template_server_app.c


template_server_app.c

```
48  /* Private defines -----------------------------------------------------------*/
49  #define TEMPERATURE_CHANGE_PERIOD   (0.1*1000*1000/CFG_TS_TICK_VAL) /*100ms*/
```

Now that the updates are coming through, we are going to create and use a time server. This is run by the RTC wakeup timer (we make use of the LSE). That means that we can create virtual timers to start, stop, pause in order to reduce energy consumption.

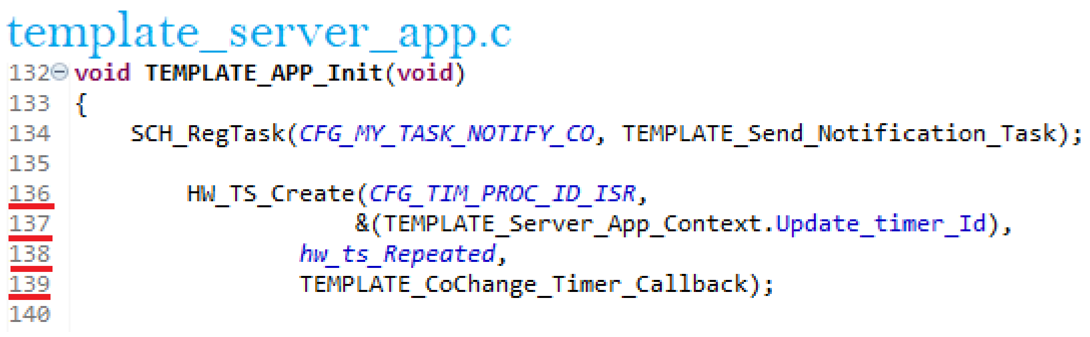

List of API

- HW_TS_Init()
- HW_TS_Create()
- HW_TS_Stop()
- HW_TS_Start()
- HW_TS_RTC_Int_AppNot()
- HW_TS_RTC_Wakeup_Handler()

- HW_TS_Delete()
- HW_TS_RTC_ReadLeftTicksToCount()
- HW_TS_RTC_CountUpdated_AppNot()

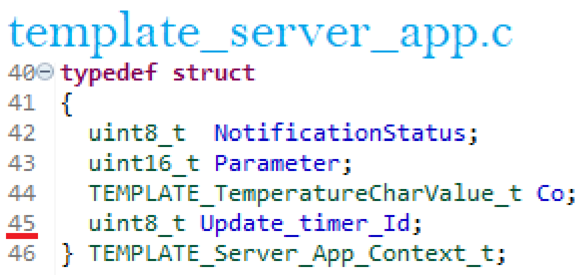
So, we are going to create a Software timer for periodic characteristic update.

We create a timer and add lines 136 till 139 in template_server_app.c


template_server_app.c

```
132 void TEMPLATE_APP_Init(void)
133 {
134     SCH_RegTask(CFG_MY_TASK_NOTIFY_CO, TEMPLATE_Send_Notification_Task);
135
136         HW_TS_Create(CFG_TIM_PROC_ID_ISR,
137                 &(TEMPLATE_Server_App_Context.Update_timer_Id),
138             hw_ts_Repeated,
139             TEMPLATE_CoChange_Timer_Callback);
140
```

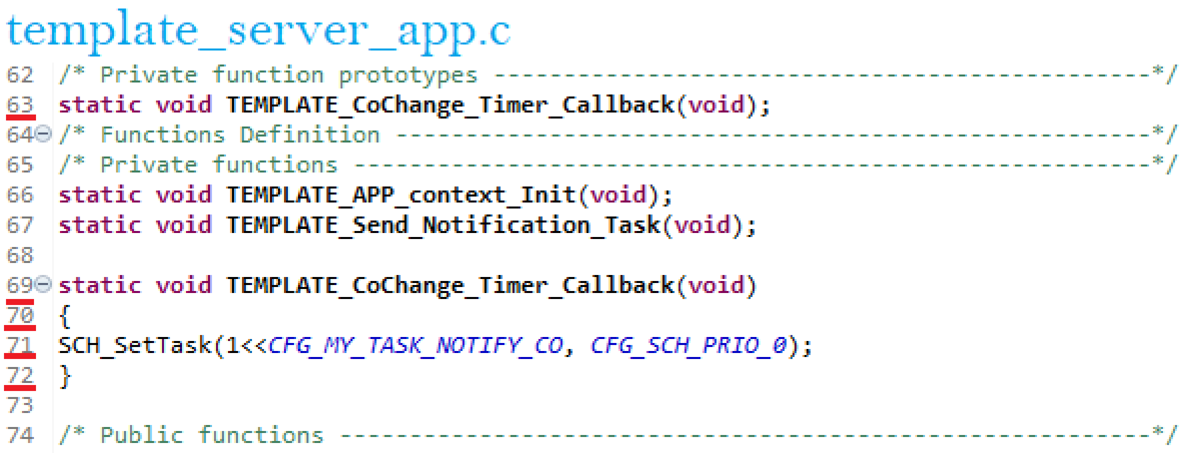
The next step for us is, to update our structure and add a software time ID. We add line 45.


template_server_app.c

```
40 typedef struct
41 {
42   uint8_t  NotificationStatus;
43   uint16_t Parameter;
44   TEMPLATE_TemperatureCharValue_t Co;
45   uint8_t Update_timer_Id;
46 } TEMPLATE_Server_App_Context_t;
```

And now we need to create the callback for the Software Timer, as to declare it and execute it.

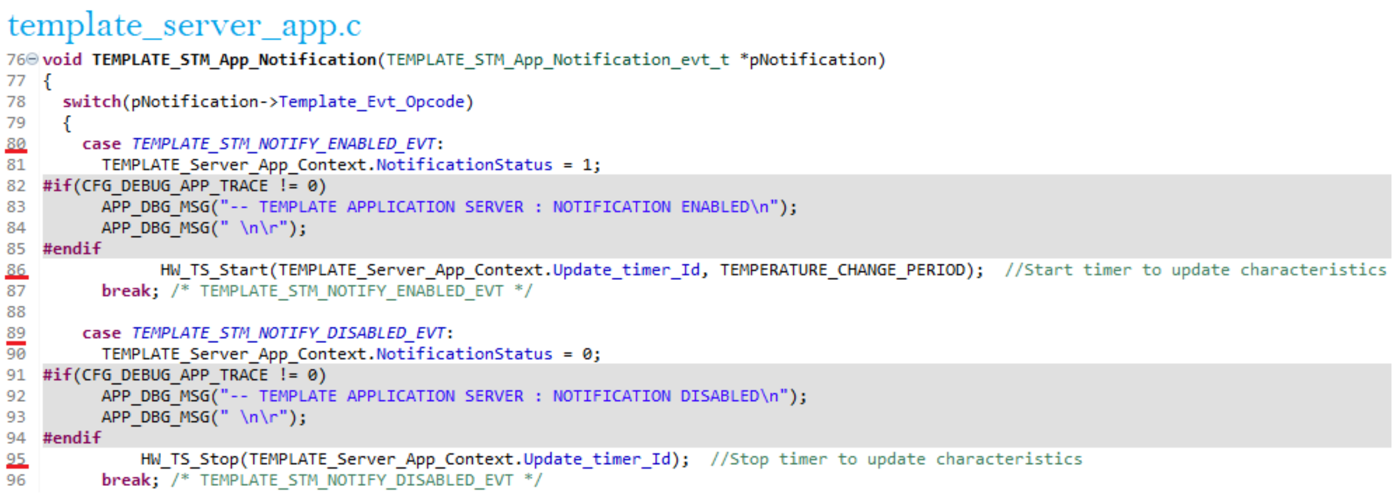
We add lines 63 and 69 till 72.



template_server_app.c

```
62  /* Private function prototypes -----------------------------------------------*/
63  static void TEMPLATE_CoChange_Timer_Callback(void);
64  /* Functions Definition ------------------------------------------------------*/
65  /* Private functions ---------------------------------------------------------*/
66  static void TEMPLATE_APP_context_Init(void);
67  static void TEMPLATE_Send_Notification_Task(void);
68
69  static void TEMPLATE_CoChange_Timer_Callback(void)
70  {
71  SCH_SetTask(1<<CFG_MY_TASK_NOTIFY_CO, CFG_SCH_PRIO_0);
72  }
73
74  /* Public functions ----------------------------------------------------------*/
```

Finally, we need to start and Stop the Software Timer. We add lines 80,86,89 and 95.



template_server_app.c

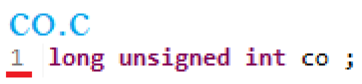
```
76  void TEMPLATE_STM_App_Notification(TEMPLATE_STM_App_Notification_evt_t *pNotification)
77  {
78    switch(pNotification->Template_Evt_Opcode)
79    {
80      case TEMPLATE_STM_NOTIFY_ENABLED_EVT:
81        TEMPLATE_Server_App_Context.NotificationStatus = 1;
82  #if(CFG_DEBUG_APP_TRACE != 0)
83        APP_DBG_MSG("-- TEMPLATE APPLICATION SERVER : NOTIFICATION ENABLED\n");
84        APP_DBG_MSG(" \n\r");
85  #endif
86            HW_TS_Start(TEMPLATE_Server_App_Context.Update_timer_Id, TEMPERATURE_CHANGE_PERIOD);  //Start timer to update characteristics
87        break; /* TEMPLATE_STM_NOTIFY_ENABLED_EVT */
88
89      case TEMPLATE_STM_NOTIFY_DISABLED_EVT:
90        TEMPLATE_Server_App_Context.NotificationStatus = 0;
91  #if(CFG_DEBUG_APP_TRACE != 0)
92        APP_DBG_MSG("-- TEMPLATE APPLICATION SERVER : NOTIFICATION DISABLED\n");
93        APP_DBG_MSG(" \n\r");
94  #endif
95          HW_TS_Stop(TEMPLATE_Server_App_Context.Update_timer_Id);  //Stop timer to update characteristics
96        break; /* TEMPLATE_STM_NOTIFY_DISABLED_EVT */
```

5.4 Transfer value Between Different Classes.

We have completed the ble and bme680 functions, but they belong into different classes.

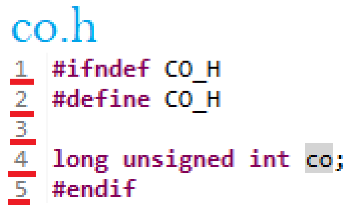
We need to be able to transfer the IAQ data from the main.c to the template_server_app.c to export them through the ble function but these classes are not connected. To do so, we will create a new class (co.c) and header (co.h).

We create these new classes and input all the lines in the next image.



co.c

```c
long unsigned int co ;
```

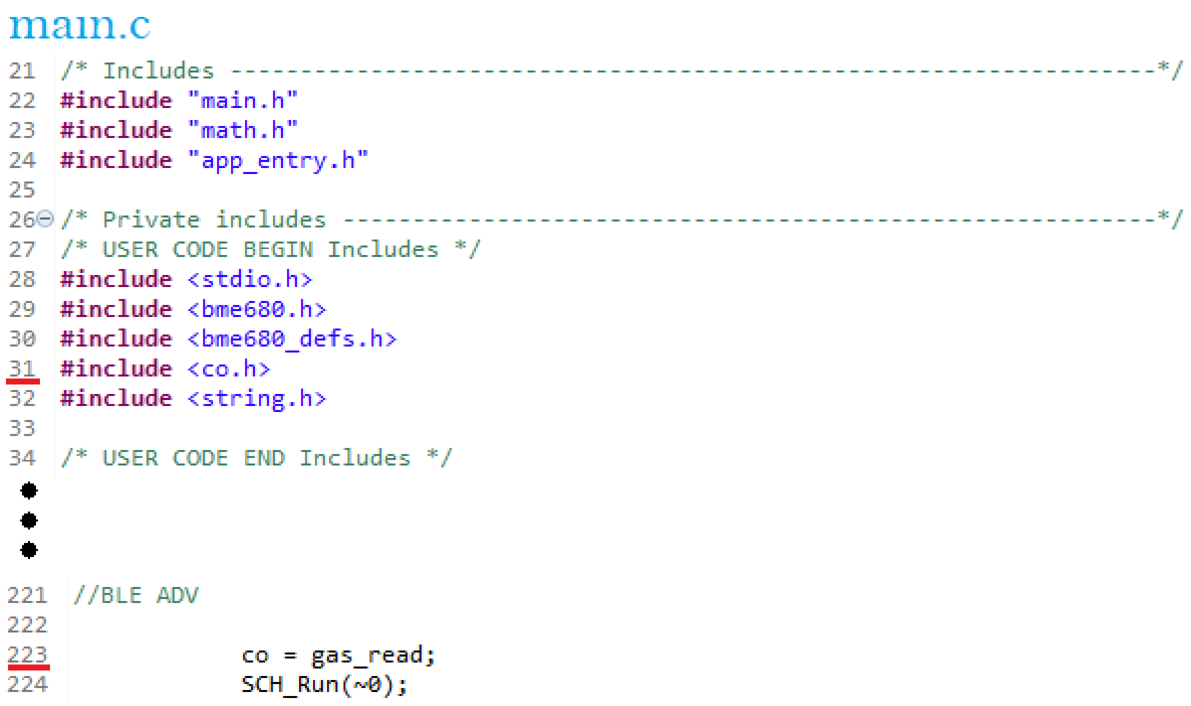


co.h

```c
#ifndef CO_H
#define CO_H

long unsigned int co;
#endif
```

We have created the new class and now we must make the connection between main.c and template_server_app.c

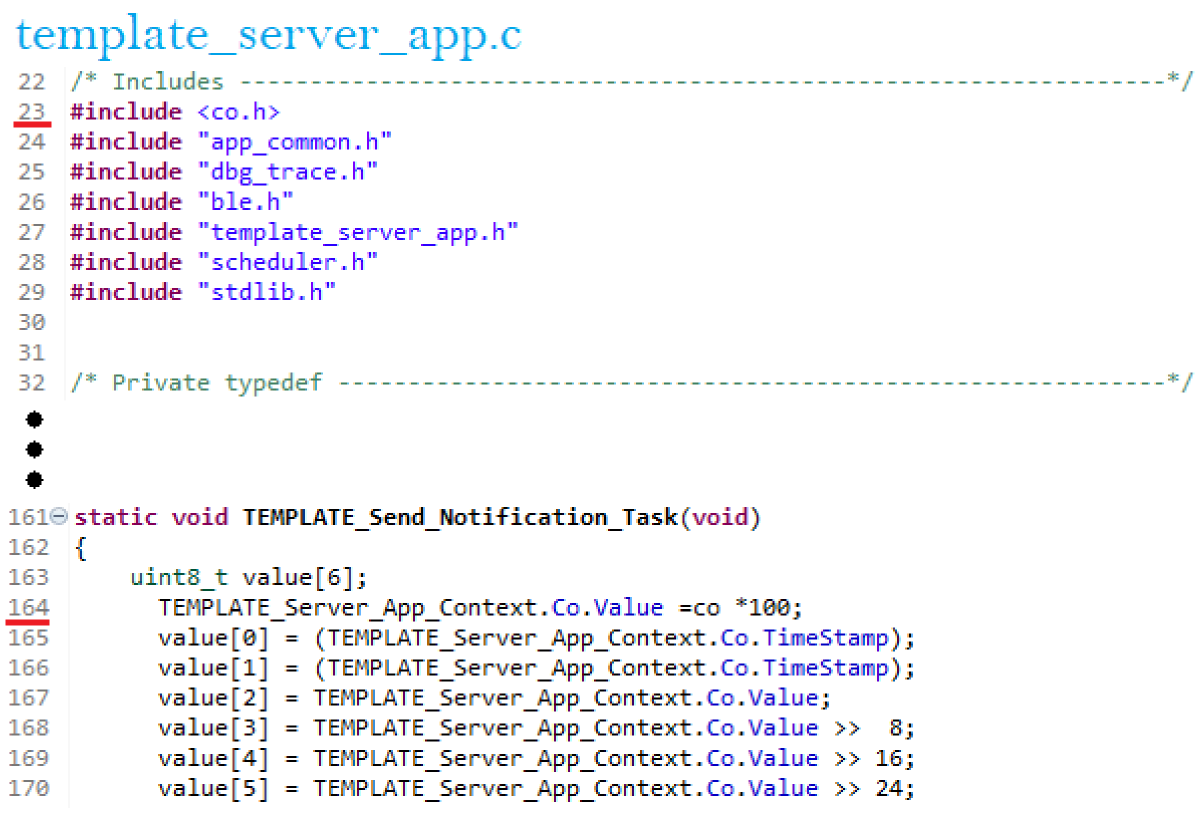
At main.c we add lines 31 and 223.



main.c

```c
21  /* Includes ------------------------------------------------------------------*/
22  #include "main.h"
23  #include "math.h"
24  #include "app_entry.h"
25
26  /* Private includes ----------------------------------------------------------*/
27  /* USER CODE BEGIN Includes */
28  #include <stdio.h>
29  #include <bme680.h>
30  #include <bme680_defs.h>
31  #include <co.h>
32  #include <string.h>
33
34  /* USER CODE END Includes */
 •
 •
 •
221 //BLE ADV
222
223             co = gas_read;
224             SCH_Run(~0);
```

At template_server_app.c we add lines 23 and 164.



template_server_app.c

```
22  /* Includes ------------------------------------------------------------------*/
23  #include <co.h>
24  #include "app_common.h"
25  #include "dbg_trace.h"
26  #include "ble.h"
27  #include "template_server_app.h"
28  #include "scheduler.h"
29  #include "stdlib.h"
30
31
32  /* Private typedef -----------------------------------------------------------*/
 ●
 ●
 ●
161 static void TEMPLATE_Send_Notification_Task(void)
162 {
163     uint8_t value[6];
164       TEMPLATE_Server_App_Context.Co.Value =co *100;
165       value[0] = (TEMPLATE_Server_App_Context.Co.TimeStamp);
166       value[1] = (TEMPLATE_Server_App_Context.Co.TimeStamp);
167       value[2] = TEMPLATE_Server_App_Context.Co.Value;
168       value[3] = TEMPLATE_Server_App_Context.Co.Value >>  8;
169       value[4] = TEMPLATE_Server_App_Context.Co.Value >> 16;
170       value[5] = TEMPLATE_Server_App_Context.Co.Value >> 24;
```

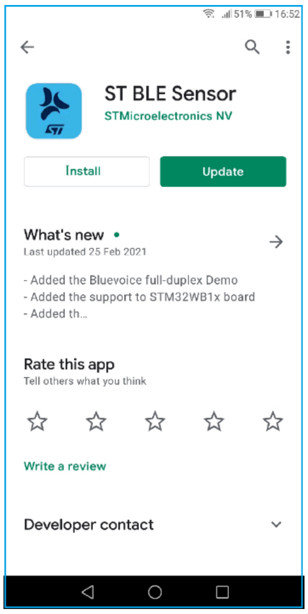
At line 164 we multiply co with 100 due to the BlueST protocol (reference at page 34).

Chapter 6: Smartphone Application

6.1 App Installation

In this chapter we will explain how we can display IAQ data remotely to our mobile phone.

We can do this with through the usage of the ST BLE Sensor app, provided free at Play Store.

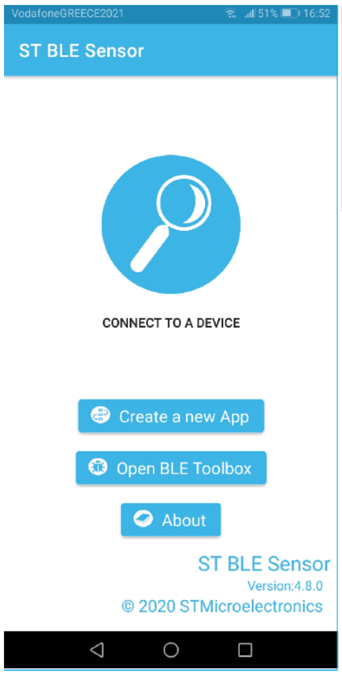


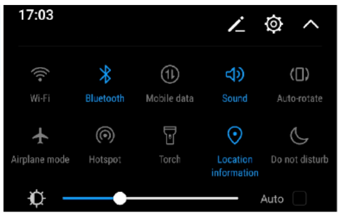
What's new
Last updated 25 Feb 2021

- Added the Bluevoice full-duplex Demo
- Added the support to STM32WB1x board
- Added th...

Once the installation is complete, the app is ready to be used. We execute the app to reach the display below.

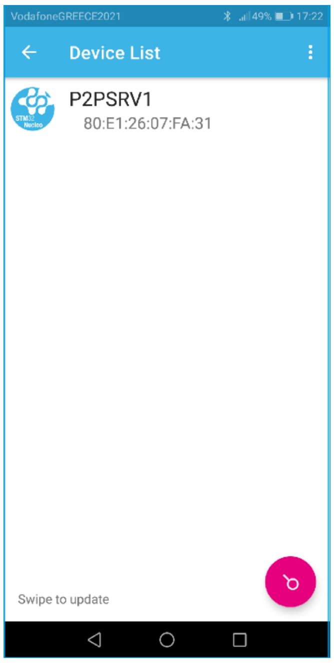
<u>Caution</u>: We must activate the Bluetooth and deactivate Wifi on our phone. Also, we must activate the Location Information.



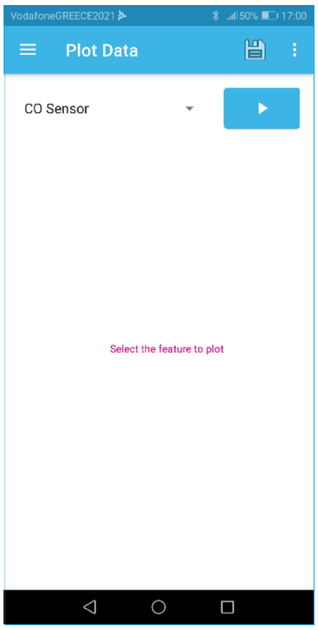


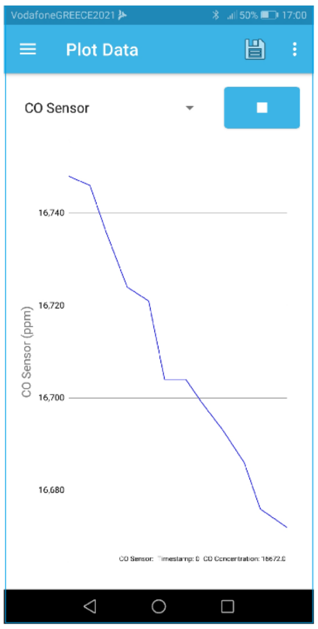
6.2 Data Collection & Visualization

Once our device is activated, we press at the “CONNECT TO A DEVICE” Button. This will open the page, displaying in the following image. Here we can view the Server Name that we gave earlier.

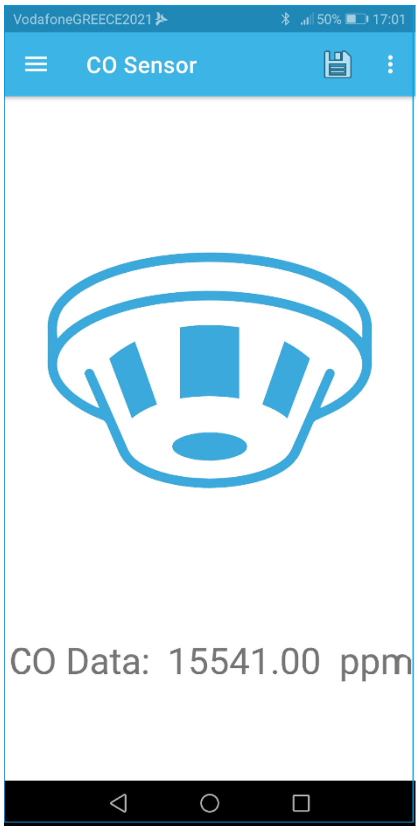


We press on the name of our device and wait for the connection to be completed. Once the connection is completed, we can swipe to see information and data from the Server. At the first we see the received data as plot. If we press the play button, data will be displayed in a plot with time.



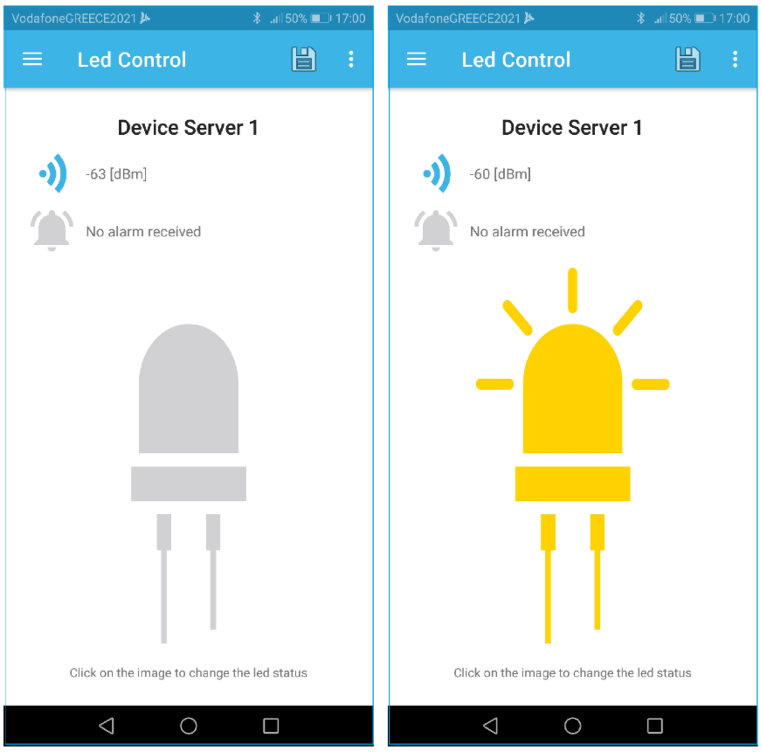


When we swipe right, we can see the current value of the IAQ that is being received.



CO Data: 15541.00 ppm

If we swipe right again, we can see the received power in dBm and we can toggle the Blue led that we have configured for debugging reasons. Once the led button has been pressed, a blue led will turn on, on our server board.



Device Server 1

-63 [dBm]

No alarm received

Click on the image to change the led status

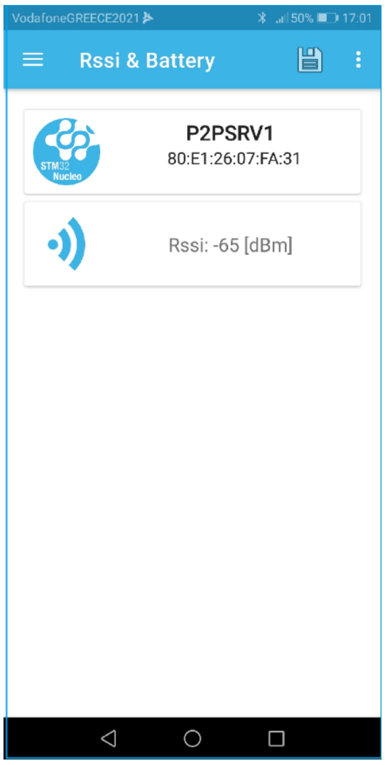
Device Server 1

-60 [dBm]

No alarm received

Click on the image to change the led status

Finally, if we swipe right again, we can view the Local name and received power of the server.



P2PSRV1
80:E1:26:07:FA:31

Rssi: -65 [dBm]

Chapter 7: Results

7.1 Final Results and Testing

Now that we have completed all the above stages of the project, our phone device is ready to receive measurements from the sensor.

The sensor that we have used (bme680) gives as an output an equivalent of CO as part of the IAQ. This is the value that we receive at our phone. That said, by performing test measurements at given space and given number of people we can determine the exact number of people in an indoor space.

With the help of Matlab and many calibrating measurements, we can reach a connection between number of people and prices of IAQ and gas resistance. This is left as future work. In the following tables we can see such measurements but in order to reach a satisfactory level, we require many more, thus future work is going to be needed.

Remember, that in order to get the IAQ we use the following function:

$$\boldsymbol{IAQ = \log(gas\ resistance) + 0.4humidity}$$

1 Person	Temp (C°)	Pressure(hPa)	Hum (%)	IAQ	Gas Resistance (ohms)
0 min	22.91	1017.77	46.96	18795	29415
10 min	23.68	1017.71	45.84	18345	32979
20 min	22.39	1017.89	46.76	18713	37865
30 min	21.68	1018.15	48.78	19523	40112
40 min	21.97	1018.41	48.37	19357	40074
50 min	22.1	1017.97	48.59	19445	40074
60 min	22.17	1017.89	49.1	19650	39619

2 Persons	Temp (C°)	Pressure (hPa)	Hum (%)	IAQ	Gas Resistance (ohms)
0 min	22.42	1008.41	55.05	22030	16723
10 min	22.65	1008.43	54.4	21771	20547
20 min	22.82	1008.53	55.21	22094	21563
30 min	22.95	1008.57	54.5	21810	22321
40 min	22.94	1008.73	54.49	21807	23110
50 min	22.97	1008.77	54.28	21723	23631
60 min	22.94	1008.79	54.28	21722	24076

The code is set to transmit only IAQ to our phone device, but by a simple change the value at line (223 of main.c) we can send any of the above measurements. Also, since the Bluetooth transmission is in advertise mode, we could collect all these measurements to a server and thus giving us the ability to store and analyze data.

Conclusions

Having completed the project we now have a fully working IoT system that gives us helpful data to calculate the exact number of people in a given indoor space.

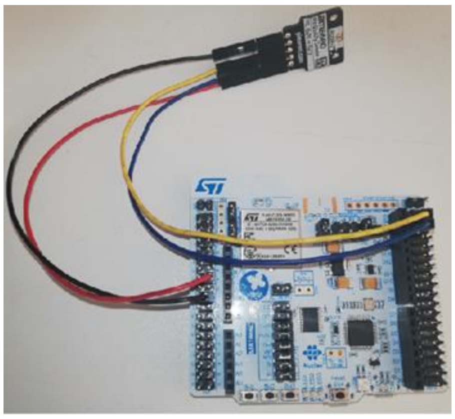
More work is needed and more tests to be executed before the device is correctly calibrated, but the system is robust and ready for use. For more accuracy in our measurements, a more expensive and accurate sensor could be used. As long as the new sensor is I2C compatible, the whole project will be ready to function with minimum changes.

Thanks to our device small size and indoor expected function, the general boxing cost, should be low. Also, since we expect the device to function indoors and for a short amount of time, the general wear and tear of the device will be at minimum, thus prolonging the expected lifetime of the device.

Moreover, the low consumption and sleep functions, further drop the usage cost and the Bluetooth addition makes the device portable.

As for the usage of the project, it could greatly help at controlling and monitoring masses of people in large and small indoor spaces. Also, the above project could help in the Health department, by controlling and keeping a low number of people, thus reducing the spread of viruses, or in the Security department, by upholding certain regulations.

In any case, the project could really help society at crowd control and to be used as a steppingstone for an even larger or future IoT applications.



Bibliography

[1] Getting Started with the Internet of Things: Connecting Sensors and Microcontrollers to the Cloud (Make: Projects) 1st Edition, Kindle Edition by Cuno Pfister

[2] Getting Started with Bluetooth Low Energy, Kevin Townsend, Charles Cufi, Akiba & Robert Davidson

[3] Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5 and 802.15.4 radio solution – Datasheet

[4] "8052-Basic Microcontrollers" by Jan Axelson 1994

[5] "The Surprising Story of the First Microprocessors". Shirriff Ken, Institute of Electrical and Electronics Engineers

[6] BME680 – Datasheet

[7] Carmine Noveillo, "Mastering STM32", https://leanpub.com/mastering-stm32

[8] https://community.st.com/docs/DOC-1413-tutorial-interfacing-a-stm32l053-discovery-with-an-i2c-sensor

[9] I^{2}C-bus specification Rev 2.1; Philips Semiconductors; January 2000

[10] I^{2}C-bus specification Rev 3; NXP Semiconductors; June 19, 2007

[11] I^{2}C-bus specification Rev 4; NXP Semiconductors; February 13, 2012

[12] I^{2}C-bus specification Rev 5; NXP Semiconductors; October 9, 2012

[13] STM32WB workshop MOOC (https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/STM32WB_workshop_MOOC.html)

[14] Getting-started-with-the-bluest-protocol-and-sdk-stmicroelectronics Datasheet