



HOGERE ZEEVAARTSCHOOL ANTWERPEN

Designing a portable and autonomous air pollution measuring instrument

Cyril Dewez

Master thesis for the obtention of
the title of
Master in Nautical Sciences

Promotor: Joeri Horvath
Co-promotor: Olivier Schalm
Academic year: 2020-2021

Foreword

This Master's thesis is one of the conditions for obtaining the Master's degree in Nautical Sciences at the Antwerp Maritime Academy.

Since the start of these studies, I have wished to study air pollution onboard merchant navy ships during my Bachelor and Master's thesis. Mr Joeri Horvath and Mr Olivier Schalm allowed me to create my own air pollution measuring instrument for that purpose. I started working on this project without any specific prior knowledge. As the design progressed, I learned to code in Python, studied sensor communications protocols, connected all the sensors individually with a breadboard, traced electronic schematics, drew my own printed circuit board with KiCad, solder connectors, designed flowcharts with draw.io, made 220V connections...

I warmly thank Mr Olivier Schalm for the opportunity to organise a trip to Cuba in this context which has been cancelled at the last minute due to the Coronavirus pandemic. I would also like to thank Mr Gustavo Carro for his help setting up my development environment. I also thank my father, Luc, for the many tips and invaluable assistance during the construction of the Seacairy case. Finally, I would like to thank the Antwerp Maritime Academy for funding the project.

Abstract

This thesis proposes the design of an instrument that measures air quality onboard merchant marine vessels. The device must fulfil the following boundary conditions: (1) be able to measure sulphur oxides, nitrogen oxides, carbon oxides, ozone, particulate matter, temperature and humidity, (2) the air supply to the sensors must go through the tubing and a pump to facilitate calibration in a later stage, (3) the instrument must be (water)tight, and (4) the instrument must be transportable.

The realization of the measuring device begins with an individual study of potential sensors. After the selection of the necessary sensors, the electronic connections have been studied, the required connectors were purchased, and the software to operate and collect data through a central computer written. Writing and installing the software on the central computer requires creating a development environment on an additional stand-alone computer. This step is studied for each sensor on a case-by-case basis. Finally, a dedicated software program synchronizes and groups the data in a shared database when all the sensors are operational. Continuously, upgrades were made, including a printed circuit board that simplifies the electric cables.

Finally, all the components of the measuring device are installed in a watertight Pelican Storm Case to form a single transportable unit. The suitcase is fitted with three aluminium plates, on which selected components are attached (e.g., pump, pipes, transformer, central unit, USB socket).

During the building process of the portable and watertight measuring device, several problems have been encountered: it is hard to find the correct component in the vast amount of possibilities on the market, the selected components are not always compatible with each other, the documentation of sensors is not always clear, errors can quickly occur with software and wires. This dissertation provides lists of components, wiring schematics, software flowcharts, and Python code examples that can be reproduced when someone wants to build a similar device.

Résumé

Ce mémoire propose la conception d'un instrument permettant la mesure de la qualité de l'air à bord des navires de la marine marchande. L'appareil se doit de répondre aux exigences suivantes : (1) être capable de mesurer les oxydes de soufre, les oxydes d'azote, les oxydes de carbone, l'ozone, les particules fines, la température et l'humidité, (2) l'arrivée d'air aux capteurs doit se faire via un système de tubes et d'une pompe de manière à faciliter la calibration dans le futur, (3) l'instrument doit être étanche (à l'eau), et (4) le tout doit être transportable.

La réalisation de l'instrument de mesure débute par une étude individuelle des capteurs. Après la sélection des senseurs nécessaires, les connections électriques sont étudiées, les connecteurs nécessaires achetés, et l'on écrit le logiciel conçu pour opérer et récupérer les données par le biais d'un ordinateur central. La rédaction et l'installation du logiciel sur l'ordinateur central nécessite la création d'un environnement de développement sur un ordinateur de bureau additionnel. Cela est étudié au cas-par-cas pour chaque capteur. Finalement, lorsque tous les capteurs sont opérationnels, un logiciel synchronise et regroupe les données dans une base de données commune. Au fur et à mesure de la conception, le système est amélioré, tel que la création d'un circuit imprimé pour remplacer tous les câbles électriques.

Finalement, tous les composants de l'instrument de mesure sont installés dans un Pelican Storm Case (valise) afin de former une seule unité transportable. La valise est composée de trois plaques en aluminium sur lesquels les composants sont fixés (par exemple, la pompe, les transformateurs, l'ordinateur central, une prise USB).

Tout au long de la conception de l'instrument de mesure, plusieurs problèmes ont été rencontrés : il est difficile de trouver les bons composants parmi les nombreuses possibilités sur le marché, les pièces sélectionnées ne sont pas toujours compatibles les unes avec les autres, et des erreurs peuvent rapidement survenir dans le logiciel et les câblages. Cette dissertation fournit des listes de composants, des schémas électriques, des algorithmes de logiciels, et des exemples de code Python, permettant à quiconque de construire un instrument de mesure similaire.

Table of content

| | |
|---|-----|
| Foreword | i |
| Abstract | iii |
| Résumé..... | v |
| Table of content..... | vii |
| Table of figures | xi |
| Table of tables | xiv |
| List of abbreviations..... | xv |
| Introduction..... | 1 |
| Chapter 1 The Sensors of the Seacairy | 3 |
| 1 E+E Elektronik EE894 CO ₂ sensor | 4 |
| 1.1 Sensor communication and wiring..... | 5 |
| 1.2 Compatible wire and sockets | 7 |
| 1.3 Software function list | 8 |
| 1.4 Software schematic | 12 |
| 1.4.1 General procedure for reading measurements | 12 |
| 1.4.2 Procedure for reading and writing bytes inside the sensor custom memory | 12 |
| 1.5 Air measurement and data reading timing..... | 14 |
| 1.6 Faced issues during the development | 16 |
| 1.6.1 Consecutive I ² C write and read | 16 |
| 1.6.2 Addition of other I ² C devices to the central computer..... | 17 |
| 1.6.3 inability to manually trigger a measurement..... | 18 |
| 1.6.4 Continuous indication of temperature error on the status byte..... | 18 |
| 1.6.5 Checksum error during measurement readings | 19 |
| 2 OPC-N3 particulate matter sensor..... | 19 |
| 2.1 Data returned by the sensor | 21 |
| 2.2 Sensor communication and wiring..... | 22 |
| 2.3 Software function list | 23 |
| 2.4 Software schematic | 29 |
| 2.4.1 SPI communication initiation..... | 29 |
| 2.4.2 Histogram reading | 29 |

| | | |
|--|---|----|
| 2.4.3 | Perform a particulate matter measurement | 30 |
| 2.5 | Faced issues..... | 33 |
| 2.5.1 | Simultaneous reading and writing of data..... | 33 |
| 2.5.2 | Sensor Slave Select line wiring | 34 |
| 2.5.3 | Frequency conflict between the OPC-N3 SPI and the GPS UART | 34 |
| 2.6 | Interference between M&C air pump and SPI communication..... | 35 |
| 2.6.1 | Isolation of the pump from the 220V line via a noise reducer..... | 37 |
| 2.6.2 | Increasing the power supply capacity | 37 |
| 2.6.3 | Addition of a rest period between starting the pump and the first communication with the sensor | 38 |
| 3 | The 4-AFE gas sensors board from Alphasense..... | 38 |
| 3.1 | Wiring of the 4-AFE board and the Analog to Digital Converter (ADC) | 39 |
| 3.2 | Software function list | 40 |
| 3.3 | Analogic signal noise reduction | 45 |
| 3.4 | Calibration..... | 45 |
| 4 | The GPS receiver..... | 45 |
| 4.1 | Wiring of the GPS receiver | 46 |
| 4.2 | Software function list | 46 |
| 4.3 | Faced issues..... | 47 |
| 4.3.1 | Frequency conflict between the OPC-N3 SPI and the GPS UART | 47 |
| 4.3.2 | Random UART port opening problem..... | 47 |
| 5 | Sensirion Mass Flow Meter | 49 |
| 6 | The RTC (real-time clock) - DS3231 | 50 |
| 6.1 | Faced issues..... | 50 |
| 6.1.1 | I ² C pull-up resistors | 50 |
| 6.1.2 | Integration on the PCB | 50 |
| Chapter 2 Combining components into a measuring device | | 53 |
| 1 | Building the device into a transportable suitcase..... | 54 |
| 1.1 | Three aluminium plates in the casing | 56 |
| 1.2 | The bottom plate | 56 |
| 1.3 | The cover plate in the case lid..... | 57 |
| 1.4 | The top plate..... | 58 |
| 1.5 | Drilling the plate to fix it on the frame | 58 |
| 2 | Connecting all sensors with tubes | 59 |
| 2.1.1 | The use of PTFE tubes..... | 62 |
| 2.1.2 | Air pump..... | 63 |
| 2.1.3 | The particulate matter sensor (OPC-N3) box..... | 64 |

| | | |
|-----------|--|----|
| 2.1.4 | The CO ₂ sensor box..... | 66 |
| 2.1.5 | The Sensirion mass flow meter | 67 |
| 2.1.6 | The Alphasense 4-AFE gas sensor | 67 |
| 2.2 | The electrical connection of all hardware components | 67 |
| 2.2.1 | Electric noise on the 220V line | 69 |
| 3 | Central computer | 70 |
| 3.1 | The Raspberry Pi | 70 |
| 3.2 | The Analog to Digital Converter | 71 |
| 3.3 | Printed Circuit Board (PCB)..... | 71 |
| 3.3.1 | General procedure for designing a PCB | 73 |
| 3.3.2 | Seacanairy wiring | 76 |
| 3.3.3 | The connection between the Analog to Digital Converter (ADC) and the custom circuit board..... | 76 |
| 3.3.4 | Tips for a successful printed circuit..... | 77 |
| Chapter 3 | Setting up the development environment on a stand-alone computer..... | 79 |
| 1 | Set up the development environment on a personal computer | 80 |
| 1.1 | Development software - PyCharm | 80 |
| 1.2 | New project creation | 81 |
| 1.3 | Git repository and GitHub account | 81 |
| 1.4 | Commit and Push files to GitHub | 83 |
| 1.5 | Libraries installation..... | 84 |
| 1.6 | Connect to the Raspberry Pi using TeamViewer | 85 |
| 1.7 | Transfer files from or to the Raspberry Pi..... | 86 |
| 2 | Set up the development environment on the Seacanairy central computer | 86 |
| 2.1 | Update the Raspberry Pi | 86 |
| 2.2 | The virtual environment on the Raspberry Pi..... | 87 |
| 2.2.1 | Create a Virtual Environment..... | 87 |
| 2.2.2 | Virtual environment activation | 87 |
| 2.2.3 | Activate the virtual environment in Thonny Python IDE | 88 |
| 2.3 | Install Python libraries on the Raspberry Pi..... | 89 |
| 2.4 | Testing code on the Raspberry Pi | 90 |
| 2.4.1 | Copy-pasting in Thonny Python IDE | 90 |
| 2.4.2 | TeamViewer File Transfer and python3 in console | 91 |
| 3 | Console tip and tricks | 92 |
| 4 | Raspberry Pi password..... | 92 |
| Chapter 4 | Software of the Seacanairy | 93 |
| 1 | Overall Seacanairy software structure | 93 |

| | | |
|-----------|--|-----|
| 2 | Information display and logging functions | 96 |
| 3 | Settings page | 98 |
| 3.1 | Choice of file format | 98 |
| 3.2 | Available settings | 99 |
| 4 | MySQL Database..... | 101 |
| 5 | Global Seacanairy script..... | 104 |
| 5.1 | Manual operation through the touchscreen..... | 106 |
| 5.2 | Autostart at boot | 106 |
| 6 | Software files and folders | 107 |
| 6.1 | List of files..... | 107 |
| | Conclusion..... | 111 |
| Annexe 1 | List of files | 113 |
| Annexe 2 | Case panels dimensions..... | 115 |
| Annexe 3 | Schematic of the Seacanairy wiring..... | 117 |
| Annexe 4 | Seacanairy PCB..... | 119 |
| Annexe 5 | CO2.py..... | 121 |
| Annexe 6 | OPCN3.py..... | 134 |
| Annexe 7 | AFE.py..... | 154 |
| Annexe 8 | GPS.py..... | 168 |
| Annexe 9 | flow.py | 176 |
| Annexe 10 | database.py | 183 |
| Annexe 11 | seacanairy_settings.yaml | 190 |
| Annexe 12 | AFE calibration | 192 |
| Annexe 13 | set_system_time.sh..... | 193 |
| Annexe 14 | Graph from the measuring device..... | 195 |
| 1 | Temperature | 195 |
| 2 | Particulate matter | 196 |
| 3 | Gas sensors | 197 |
| 4 | Air flow | 198 |
| 5 | SO2 peak when a nearby lawn tractor passes | 199 |
| | Bibliography..... | 200 |

Table of figures

| | | |
|-----------|---|----|
| Figure 1 | Picture of the CO2 sensor..... | 5 |
| Figure 2 | I ² C communication sentence..... | 7 |
| Figure 3 | CO2 sensor connectors | 7 |
| Figure 4 | General procedure for reading CO ₂ sensor measurements flowchart..... | 13 |
| Figure 5 | Reading and writing inside sensor custom memory flowchart | 14 |
| Figure 6 | CO ₂ sensor sampling timing | 16 |
| Figure 7 | Schematic of I ² C write and read sentences without any STOP bit in between | 17 |
| Figure 8 | I ² C write and read sentences without any STOP bit in between (Python code)..... | 17 |
| Figure 9 | Pictures of the OPC-N3..... | 20 |
| Figure 10 | Schématic of the connection of multiple SPI devices to the same Master | 23 |
| Figure 11 | Flowchart of the SPI communication initiation..... | 31 |
| Figure 12 | Flowchart of the histogram reading | 32 |
| Figure 13 | Flowchart of getting data from the OPC-N3 sensor..... | 33 |
| Figure 14 | Power supply of the central computer (Raspberry Pi and printed circuit) and 1 Farat capacitance | 37 |
| Figure 15 | Schematic representation of the Alphasense 4-AFE wiring | 40 |
| Figure 16 | Velleman VMA430 and U-BLOX NEO-7M chip | 46 |
| Figure 17 | ls -l /dev* on Raspberry Pi, UART configuration | 48 |
| Figure 18 | Switching UARTs on the Raspberry Pi (config.txt)..... | 49 |
| Figure 19 | RTC DS3231 chip..... | 50 |
| Figure 20 | Relocation of the DS3231 socket to solve the PCB design problem..... | 51 |
| Figure 21 | The case of the Seacanairy..... | 54 |
| Figure 22 | Pelican Storm Case iM2720 before/after | 56 |
| Figure 23 | Bottom plate fixing bolts (and the four bolts) | 57 |
| Figure 24 | Picture of the bottom plate and its components | 57 |
| Figure 25 | Cover plate (in the case lid), back and front side | 58 |
| Figure 26 | Top plate, back and front side..... | 58 |

| | | |
|-----------|--|----|
| Figure 27 | Schematic of the inboard piping system | 60 |
| Figure 28 | Picture of the piping system inside the Pelican case..... | 60 |
| Figure 29 | Air pump in its initial situation (on the left), unbolted, and rotated (on the right) 64 | |
| Figure 30 | Operation of the needle valve of the M&C air pump | 64 |
| Figure 31 | The OPC-N3 box..... | 65 |
| Figure 32 | Connection of the tube system to the OPC-N3 via a Swagelok connector and four threaded rods..... | 65 |
| Figure 33 | Fixing the OPC-N3 box to the bottom case panel | 66 |
| Figure 34 | The CO ₂ box..... | 67 |
| Figure 35 | Shaping of the M&C connector and assembly of waterproof connectors | 67 |
| Figure 36 | Picture of the 220V derivation box (on the left), and the Tokin noise filter (on the right) | 69 |
| Figure 37 | Schematic of the wiring of the Tokin noise filter on the M&C air pump..... | 70 |
| Figure 38 | Central computer unit (from bottom to top: Raspberry Pi, Pi16-ADC, custom printed circuit board) | 70 |
| Figure 39 | Picture of the Raspberry Pi 3B+ (on the left) and the PI-16ADC (on the right). | 71 |
| Figure 40 | Overview of the wiring of the first prototype (on the left side) and overview of the wiring of a similar system using the PCB-board (on the right side) | 73 |
| Figure 41 | Comparison of the symbol on the schematic with the footprint on the printed circuit board | 74 |
| Figure 42 | Positioning of the footprints and tracing of the electric lines | 75 |
| Figure 43 | Printed circuit as supplied by Eurocircuits | 75 |
| Figure 44 | Welding the connectors on the custom PCB | 76 |
| Figure 45 | Connection between the printed circuit board and the ADC..... | 77 |
| Figure 46 | Female header on the ADC and male header on the printed circuit board | 77 |
| Figure 47 | PyCharm screenshot..... | 80 |
| Figure 48 | Create a new project in PyCharm | 81 |
| Figure 49 | Git incorporation to PyCharm..... | 82 |
| Figure 50 | Log in GitHub using PyCharm | 82 |
| Figure 51 | Create Git repository through PyCharm | 83 |
| Figure 52 | Commit and Push changes to GitHub | 84 |
| Figure 53 | GitHub repository example..... | 84 |

| | | |
|-----------|--|-----|
| Figure 54 | Install libraries on PyCharm..... | 85 |
| Figure 55 | File transfer from/to the Raspberry Pi..... | 86 |
| Figure 56 | Activate the virtual environment on the Raspberry Pi..... | 88 |
| Figure 57 | Opening Thonny Python IDE..... | 89 |
| Figure 58 | Virtual environment in Thonny Python IDE..... | 89 |
| Figure 59 | Copy-pasting code from PC to Thonny Python IDE | 91 |
| Figure 60 | Testing code using file transfer and console | 92 |
| Figure 61 | Seacanairy software structure | 94 |
| Figure 62 | Importation in Python example | 95 |
| Figure 63 | General Python script layout..... | 96 |
| Figure 64 | Logging flowchart..... | 98 |
| Figure 65 | Visual comparison between JSON and YAML..... | 99 |
| Figure 66 | MySQL connection process flowchart | 102 |
| Figure 67 | Database software flowchart | 103 |
| Figure 68 | Display of the data stored in the MySQL database | 104 |
| Figure 69 | Flowchart showing Seacanairy software functioning..... | 105 |
| Figure 70 | Welcome screen of the Seacanairy (shown on the touchscreen) | 106 |
| Figure 71 | Seacanairy service status..... | 107 |
| Figure 72 | Files used by the Raspberry Pi for the proper execution of the software..... | 109 |
| Figure 73 | Lid and Base panel plan | 115 |
| Figure 74 | Bottom panel plan | 116 |
| Figure 75 | Seacanairy electronic and electric schematic..... | 117 |
| Figure 76 | Seacanairy PCB version 2.0 (current version)..... | 119 |
| Figure 77 | Seacanairy PCB version 3.0 (RTC DS3231 corrected)..... | 120 |
| Figure 78 | Graph of the temperature measured in a garden in the countryside..... | 195 |
| Figure 79 | Graph of particulate matter sampled in a garden in the countryside..... | 196 |
| Figure 80 | Graph of gas concentration in a garden in the countryside..... | 197 |
| Figure 81 | Graph of the flow rate measurement of the Seacanairy while sampling in a garden in the countryside..... | 198 |
| Figure 82 | Graph of gas concentration measurements on a terrace in the countryside | 199 |

Table of tables

| | | |
|----------|--|-----|
| Table 1 | List of Seacanairy sensors, what they measure, and their communication protocol. | 4 |
| Table 2 | Epluse E+E Elektronik CO2 sensor characteristics | 4 |
| Table 3 | CO ₂ Sensor tensions and resistors manufacturer's recommendations | 6 |
| Table 4 | CO2.py list of functions | 9 |
| Table 5 | OPC-N3 sensor characteristics | 20 |
| Table 6 | Compatible sockets with the OPC-N3 | 22 |
| Table 7 | OPCN3.py list of functions | 25 |
| Table 8 | Inventory of the Alphasense gas sensor | 39 |
| Table 9 | OPCN3.py list of functions | 41 |
| Table 10 | GPS.py function list | 46 |
| Table 11 | Inventory of the components needed to build the casing | 55 |
| Table 12 | Inventory of the piping system | 61 |
| Table 13 | Electrical connections for power supply | 68 |
| Table 14 | Inventory of the Central Computer | 71 |
| Table 15 | Inventory of the printed circuit board | 72 |
| Table 16 | Raspberry Pi TeamViewer ID and Password | 86 |
| Table 17 | pip3 function list | 90 |
| Table 18 | Tip and tricks console | 92 |
| Table 19 | Raspberry Pi username and password | 92 |
| Table 20 | Functions to manage the Seacanairy service for autostart after boot | 106 |

List of abbreviations

| | |
|-----------------------|---|
| .py | Python filename extension |
| ADC | Analogue to digital converter |
| AFE | Analogue Front End |
| CS | Chip Select |
| CS | Clock stretching |
| GND | Ground |
| GPIO | General Purpose Input/Output |
| GPIO | General Purpose Input/Output |
| HAT | Hardware Attached on Top |
| I²C | = IIC: Inter-Integrated Circuit |
| IDE | Integrated Development Environment |
| MISO | Master In Slave Out |
| MOSI | Master Out Slave In |
| NL | Normal Liter: gas volume unit at standard pressure and temperature conditions (0°C, 1 bar) |
| OPC | Optical Particulate Counter |
| PC | Personal Computer |
| PCB | Printed Circuit Board |
| RX | Receive |
| SCCM | Standard cubic centimetre per minute: flow rate of a gas at standard pressure and temperature conditions (0°C, 1 bar) |
| SCL | Serial Clock |
| SCLK | Serial Clock |
| SDA | Serial Data Line |
| SLH | Standard litre per hour: flow rate of a gas at standard pressure and temperature conditions (0°C, 1 bar) |
| SLM | Standard litre per minute: flow rate of a gas at standard pressure and temperature conditions (0°C, 1 bar) |
| TX | Transmission |
| UART | Universal asynchronous receiver transmitter |
| USB | Universal Serial Bus |
| SPI | Serial Peripheral Interface |

| | |
|-------------|-------------------------------------|
| NDIR | Nondispersive infrared (technology) |
| ACK | Acknowledge |
| NACK | Not acknowledged |

Introduction

A good breath of fresh sea air inspires many people to travel to the sea or go on a cruise trip. Unfortunately, ships emit substantial amounts of air pollutants. Although the contaminations are invisible, colourless and odourless, sea air is not as pure as it used to be. Suppose the wind and ventilation direct these pollutants towards living areas. In that case, crew and passengers are exposed to sulphur oxides, particulate matter, or nitrogen oxides. It is hard to evaluate if the air quality in the ship's surroundings is always good or bad or if pollution occurs only at specific times. Also, one wonders what the effect might be for a cocktail of pollutants.

These questions can only be answered by thoroughly analyzing the air quality in and around ships. For this purpose, equipment is required to measure the concentration of several ships' specific pollutants in real-time. Many crowd-sourced science projects suggest the creation of such kinds of devices using low-cost components. However, the research question can only be answered using reliable, calibrated, and relatively easy systems. This thesis aims to propose the design of such a device: the Seacanairy.

The first chapter introduces the selected sensors, which are the heart of the Seacanairy. Each sensor connected to the central computer requires a specific wiring and electrical connection, communication protocol, and proper software to ensure correct operation and data retrieval. Along the way of designing the Seacanairy, several problems (e.g., interferences) were encountered with various sensors, requiring research to find a proper solution. A dedicated subchapter inventories the troubles encountered.

The second chapter explains how all the required components are combined into one single instrument inside a solid suitcase. The most challenging part consists of finding the necessary components on the market, considering compatibility with the other elements. An overview of the components used and their suppliers are provided to build a similar device for every embedded system. A tube connects all the sensors so that the same air passes through the sensors one after the other. During this stage, several problems were met, such as the shape of

the particulate matter sensor (OPC-N3), which does not allow an easy coupling with the tubes. In addition, all these sensors are connected to a central computer through electrical wiring, leading to interferences between different devices.

The fourth chapter explains setting up a development environment on a stand-alone computer to work on the Seacairy software. Finally, the fifth chapter covers the Seacairy software that synchronizes all the sensors, manages settings, stores measurements into a database, and interacts with the operator. There is also a connection between the Seacairy measuring tool and the cloud to access the data remotely in real-time.

Chapter 1

The Sensors of the Seacanairy

The Seacanairy is equipped with several sensors to measure the environmental parameters necessary to determine the air quality, such as gas sensors, particulate matter, temperature, pressure, and relative humidity. This chapter covers a series of points for each sensor employed within the Seacanairy. First, each device is designed to communicate with a computer via a specific communication protocol, such as UART, I²C, SPI or analogue. Each protocol has its own technical characteristics and therefore requires between 2 and 5 cables. Generally, on the sensor side, the connection is made via a female socket. The right compatible wire should be purchased as soon as possible to run some tests as the software is written. Second, the manufacturer's documentation gives instructions on how to communicate with the sensor. Those papers are often complex and incomplete. Making the correct electrical connection and performing the proper communication operations through custom software requires hours of research and many trials and errors. Every binary data sent by the sensor must be correctly decrypted and converted by the central informatics unit in readable values, i.e. concentrations, temperatures, pressures... After extensive research and many retrials, the software performs the processes smoothly and manage automatically every step to control the sensor and retrieve its data. For each sensor, a table lists all the software functions. Some flowcharts demonstrate the logic followed by the software. Finally, the last point explains the issues encountered, their causes (or hypothesis) and solutions. Table 1 lists all the sensors on board the Seacanairy.

Table 1 List of Seacairy sensors, what they measure, and their communication protocol

Source: own work

| Piece No. | Name | Parameters | Communication |
|-----------|---|--|-------------------------------------|
| 1 | E+E Elektronik EE894 CO ₂ sensor | CO ₂ , temperature, relative humidity, atmospheric pressure | I ² C (E2 ¹) |
| 2 | Alphasense OPC-N3 | Particulate Matter | Serial |
| 3 | Alphasense 4-AFE gas sensors | NO ₂ , O ₃ , SO ₂ , CO, temperature | Analog |
| 4 | GPS | Latitude, Longitude, speed, heading, time... | UART |
| 5 | Sensirion Mass Flow Meter 4100 | Air flow | I ² C |
| 6 | Real Time Clock | Time | I ² C |

1 E+E Elektronik EE894 CO₂ sensor

The CO₂ sensor (see Figure 1) is an “Epluse E+E Elektronik EE894-HV2PCB8E25 Compact”, which measures CO₂, temperature, humidity, and ambient pressure for changing environmental conditions. The specifications of this sensor can be found in Table 2.

Table 2 Epluse E+E Electronic CO₂ sensor characteristics

Source: adapted from the official documentation [11]

| | CO ₂ | Temperature | Relative Humidity | Atmospheric pressure |
|-------------------------------------|---|-----------------------|-------------------|---|
| Units | <i>ppm</i> | <i>°C</i> | <i>%RH</i> | <i>mbar</i> |
| Range | 0 → 5000 <i>ppm</i> | −40 → 60 <i>°C</i> | 0 → 95 <i>%RH</i> | 700 → 1100 <i>mbar</i> (non-condensing) |
| Accuracy (at 25°C and 1013 mbar) | ± 50 <i>ppm</i> + 3% of the measured value | ± 0.5 <i>°C</i> | ± 3 <i>% RH</i> | ± 2 <i>mbar</i> (from 20 to 80 <i>% RH</i>) |
| Calibration | Every five years | - | - | - |
| Response time | 105 seconds with measured data averaging 60 seconds with an instant data reading | - | - | - |

¹ Proprietary protocol.

The sensor relies on dual-wavelength NDIR (nondispersive infrared) technology to get long-term stable CO₂ readings. An infrared source with a specific wavelength and frequency irradiates the gas chamber. Each molecule's atoms have their resonance frequency in function of their mass. Therefore, CO₂ molecules in the sampling space will resonate and vibrate at a known frequency which is the one used by the infrared source. A detector measures the residual infrared energy behind an optical filter at the opposite side of the infrared light source. The higher the CO₂ concentration is, the more molecules will absorb the infrared light, and the less radiation will be detected. Then, the sensor firmware converts this reading to a CO₂ concentration by using the Lambert-Beer Law and applying compensation for temperature, atmospheric pressure and humidity to increase the reading's accuracy [6,11]. Fortunately, the sensor does all of these calculations itself and returns the concentration directly.

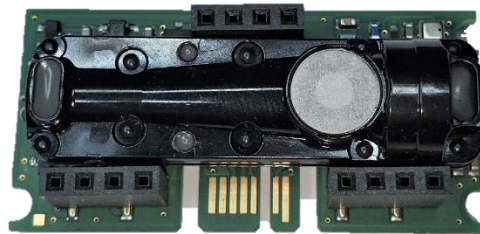


Figure 1 Picture of the CO₂ sensor

Source: own work

1.1 Sensor communication and wiring

The CO₂ sensor requires two wires for power (pin 2 connected to the ground GND and pin 1 connected to 5V of the Raspberry Pi). Sensor communication relies on the proprietary E2 protocol, which is derived from the I²C and SMBus protocol. I²C communication requires two wires: the SDA (Serial Data line) on pin 3 and the SCK (Serial Clock) on pin 2. The pull up resistors connect the two communication lines with the 3.3 Volts of the Raspberry Pi to gives a high state on the SDA and SCK in standby mode. When data is transferred, the CO₂ chip connects and disconnects the lines to the ground to tune the voltage according to the I²C standard protocol. Annexe 3 on page 117 shows the CO₂ sensor wiring when connected alone to the Raspberry Pi, and Table 3 indicates additional information concerning the wiring.

Table 3 CO₂ Sensor tensions and resistors manufacturer's recommendations

Source: own work and manufacturer documentation [33]

| | Value recommended by the manufacturer | Value used in the Seacanairy |
|--|---------------------------------------|------------------------------|
| Bus High Voltage | 3.3 → 5.2V | 5V |
| Pull-up resistor (R1 + R2, R3 + R4) | 4.7 → 100 kΩ | 20 kΩ |

All I²C communication is based on the same method. Also, I²C compatible devices are wired in parallel, and every device has its own I²C address, defined by the manufacturer. All communications are composed of sentences, on request of the user software. To start, the Master (in our case, the central computer – see point 0 in Chapter 1 on page 3) send a START bit followed by the device address byte it wishes to contact. The last bit indicates whether the Master wishes to write (from Master to Slave) or read (the opposite). Then, the concerned device replies with an ACK (acknowledge) or NACK (the opposite). Afterwards, bytes (composed of eight bits) follow as written in the software, separated by ACK or NACK bits. An ACK indicates successful reception of the last transmitted byte, while a NACK bit indicates an error when receiving the last byte, a complete memory, or the last byte of a read sentence. Finally, the Master completes the sentence with a STOP bit [15]. Figure 2 summarizes a regular I²C communication [13]. For this Raspberry Pi, the `smbus2` Python library must be installed. The sensor requires the Master to support clock stretching. As seen in Figure 2, each bit is transferred during clock rises. If the sensor firmware is not ready to receive the next bit, it will keep the clock line down until it is ready to read the next bit. In that case, the Master waits for the sensor to release the clock line to transfer the next bit.

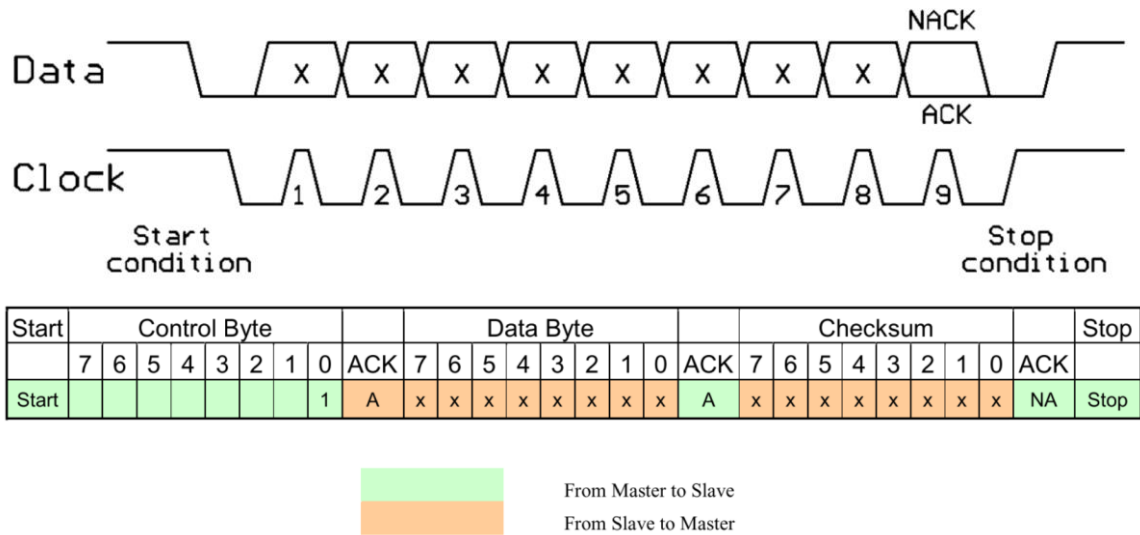


Figure 2 I²C communication sentence

Source: adapted from the manufacturer's documentation about I²C communication [13]

1.2 Compatible wire and sockets

The sensor has two different connectors (see Figure 3). The first one is a standard 2.54 mm pitch female header, which is the one used on the Seacanairy. Nevertheless, another male connection is available on the side of the sensor. The compatible female socket is a 1.00 mm Mini Edge Card with reference Samtec MEC1-108-02-S-D-A. No connections should be made to the other available headers² (see Annexe 3 on page 117 and Figure 3).

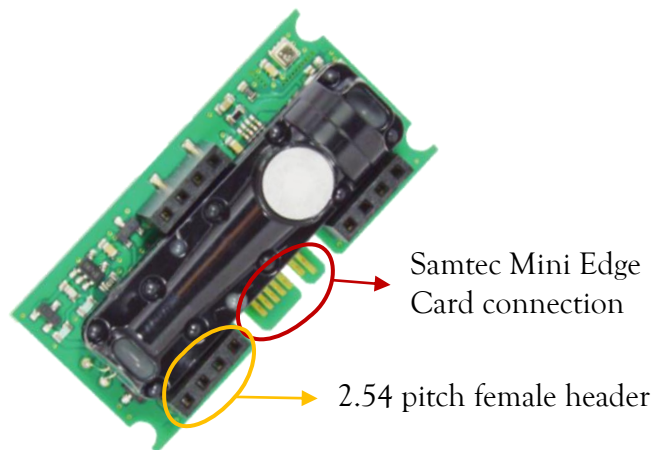


Figure 3 CO₂ sensor connectors

Source: own work and official documentation [11]

² Manufacturer's recommendation.

1.3 Software function list

In order to operate the sensor correctly from the central computer, Python code has been written based on the manufacturer's official documentation and I²C examples from the internet [8,9,10,11,18,31,33]. After hours of trial and error, the software performs the communication properly, the data verification, the conversion of the transmitted bytes into measurements, and the change of parameters within the sensor. Table 4 list all the functions of the CO₂ software. Note that `get_data()` is the final function that performs all the necessary operations to obtain all the measurements easily. A copy of the Python file (`CO2.py`) is available in Annexe 5 on page 121.

Table 4 CO2.py list of functions

Source: own work, with the help of the manufacturer's documentation [8,9,13]

| Function | Goal | Argument | Return |
|---|--|---|---|
| loading_bar (name, delay) | Show a loading bar on the screen for a certain amount of time. Make the user understand the software is doing/waiting for something | name: Text to show on the left of the loading bar (waiting, sampling...) delay: the amount of time the system is waiting (seconds) | Nothing |
| digest (buf) | Calculate the CRC8 checksum (based on the CO2 documentation example) | buf: List of bytes to digest [bytes to digest] | Calculated checksum |
| check (checksum, data) | Check that the data transmitted are correct using the data and the given checksum | checksum: Checksum given by the sensor (see sensor doc) data: List of bytes transmitted by the sensor before the checksum (see sensor doc) | True if the data are correct, False if not |
| Status (print_information=True) | Read the status byte of the CO2 sensor !! It will trigger a new measurement if the previous one is older than 10 seconds | print_information: Optional: False to hide the screen messages | True if the last measurement is OK, False if NOK |
| getRHT () ³ | Read the last Temperature and Relative Humidity measured, process the bytes, check a checksum, convert in °C and %RH | | Dictionary with the following items {"RH", "temperature"} |

³ Refer to Figure 4 on page 50 for detailed flowchart.

| Function | Goal | Argument | Return |
|--|---|---|---|
| getCO2P() ³ | Read the last CO2 instant, CO2 average and pressure measurements, process the bytes, check checksum, convert in hPa and ppm | | Dictionary containing the following items {"average", "instant", "pressure"} |
| get_data() | Get all the available data from the CO2 sensor (CO2 instant/average, pressure, temperature, humidity) | | Dictionary containing the following items { "pressure", "temperature", "CO2 average", "CO2 instant", "relative humidity"} |
| internal_timestamp (new_timestamp=None) | Read the internal sampling period of the CO2 sensor. To change the value, write it between the brackets (in seconds) | new_timestamp : None or empty to read, new value in seconds to change it. | Actual internal sampling period of the sensor |
| trigger_measurement (force=False) | Request a new CO2, t°, pressure and RH measurement IF the previous one is older than 10 seconds. Force to avoid the previous 10 second's condition. Same function as 'status()' | force : True to apply the function two consecutive times to be sure that the sensor is well synchronized with the Seacanairy; False to apply it once (during the main loop of the Seacanairy, for example) | True or False if status if OK or NOK |
| read_internal_calibration (item) | Read the internal calibration of a particular sensor item | item : indicate which internal calibration to read: 'relative humidity', 'temperature', 'pressure', 'CO2', 'all' | A list containing the calibration settings [offset, gain, lower_limit, upper_limit] |
| read_from_custom_memory (index, number_of_bytes) ⁴ | Read bytes from specified custom memory address in the CO2 sensor internal memory | index : index of the data to be read (see sensor doc) number_of_bytes : number of bytes to read (see sensor doc) | list[bytes] from right to left |

⁴ Refer to Figure 5 on page 22 for detailed flowchart.

| Function | Goal | Argument | Return |
|--|---|---|-----------------------------------|
| <code>write_to_custom_memory (index, *bytes_to_write)⁴</code> | Write data to a custom memory address in the CO2 sensor internal memory | index: index of the customer memory to write (see sensor doc) bytes_to_write: unlimited amount of bytes to write into the internal custom memory at index (see sensor doc) | True (Success) or False (Failure) |

1.4 Software schematic

The purpose of the following flow charts is to illustrate graphically the various stages conducted by the Python software during the execution of various functions. The steps shown are the result of extensive research based on the manufacturer's documentation, internet examples, `smbus2` library documentation, and trial and error.

1.4.1 *General procedure for reading measurements*

Figure 4 on page 13 is a flow chart graphically representing all the operations automatically performed by the `getCO2P()`, `getRHT()`, and `get_data()` functions. The corrugated rectangle in the top right of the flow chart represents the Seacairy settings file (see point 3 on page 98). In the centre of the flow chart, the large rectangle represents the I²C communication. On request, the sensor sends the bytes containing the measurements, followed by the result of a known calculation based on the bytes transmitted, called the sensor checksum. Then, the central computer performs the same calculation with the received bytes and compares its result with the sensor's result. If the two checksums are identical, then the bytes are the same, and the transmission is successful. However, if the checksums are not identical, it means that bytes received by the central computer are corrupted.

1.4.2 *Procedure for reading and writing bytes inside the sensor custom memory*

The CO₂ sensor has an internal memory holding a series of numbered bytes. Each byte corresponds to a specific setting, such as the measurement period, calibration, or sensor status. This memory is accessible by the user and keeps the settings even in a power supply interruption. In order to use the maximum of available sensor functionality, it is, therefore, necessary to be able to read and change the content of the sensor's memory. Figure 5 on page 14 is a flow chart representing the procedure followed by the software for reading and writing bytes inside the sensor memory. Functions `read_from_custom_memory(index, number_of_bytes)` (for reading bytes from the sensor to the central computer) and `write_to_custom_memory(index, *bytes_to_write)` (for writing bytes from the central computer to the sensor memory) automatically performs all steps shown in the flowchart.

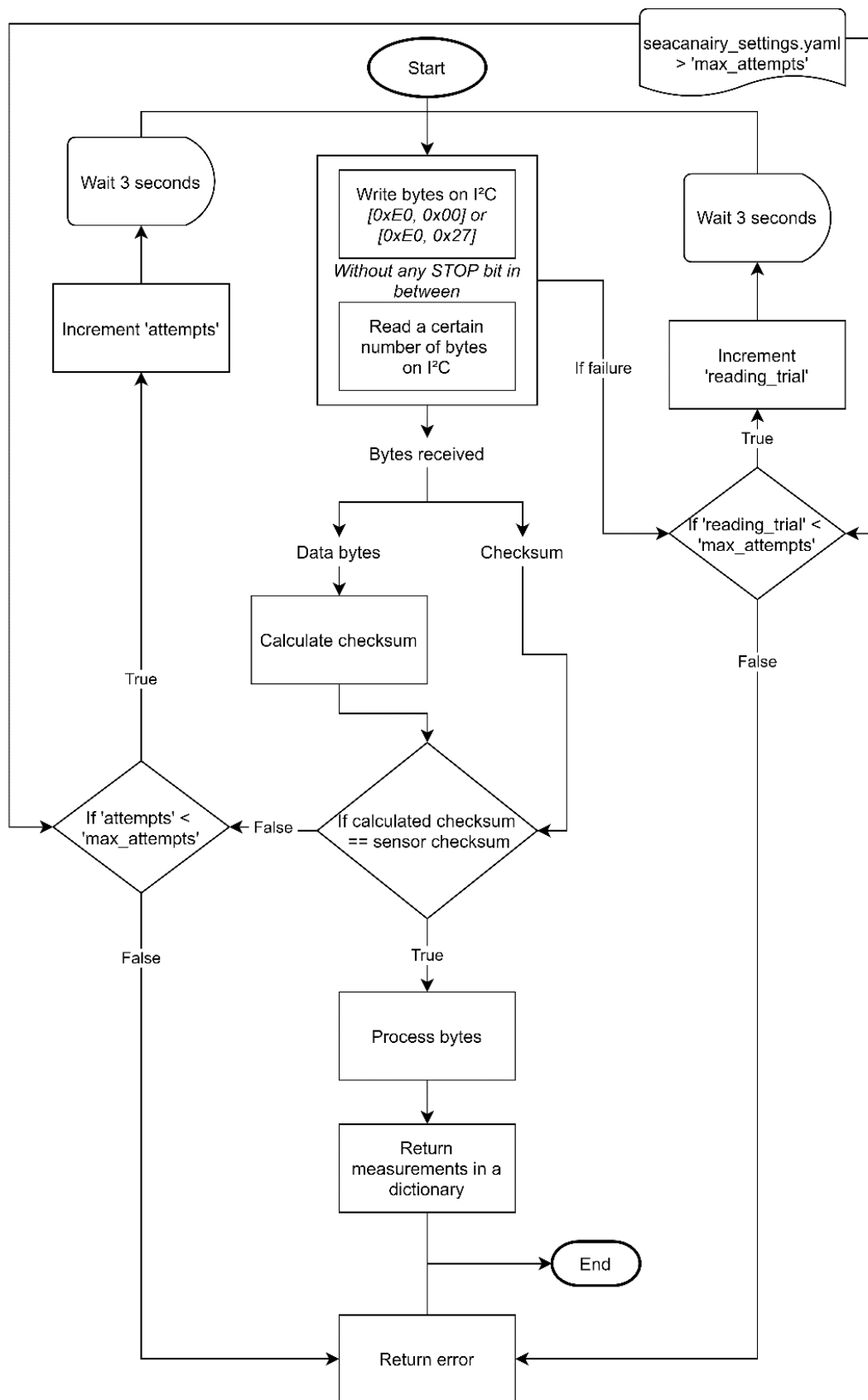


Figure 4 General procedure for reading CO₂ sensor measurements flowchart

Source: own work, based on manufacturer's documentation instructions [8,9,10,13]

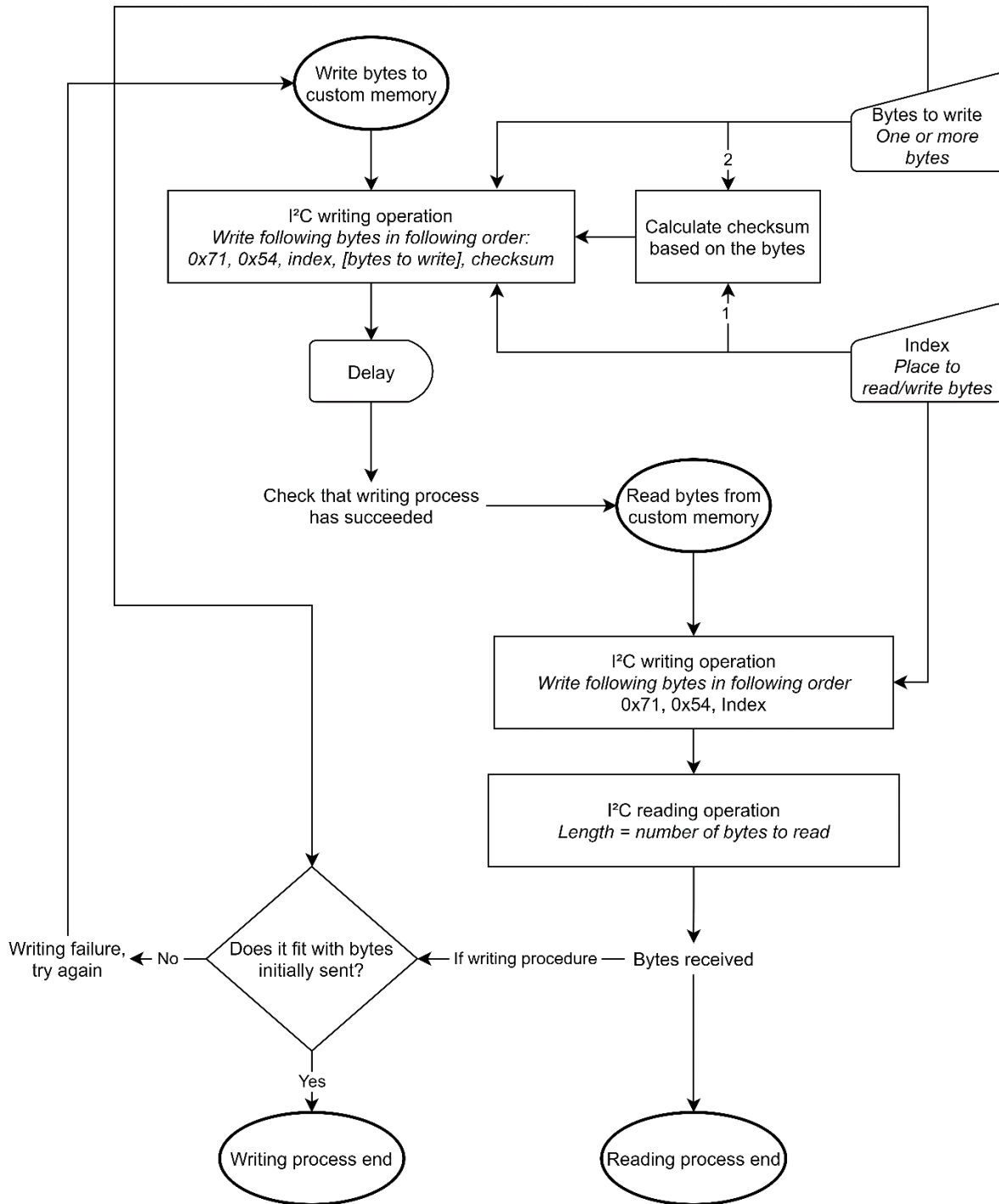


Figure 5 Reading and writing inside sensor custom memory flowchart

Source: own work based on the manufacturer's documentation [8,9]

1.5 Air measurement and data reading timing

The CO₂ sensor is designed to take measurements autonomously at a constant sampling time. Function `internal_timestamp()` writes the desired sampling time in seconds (a number between 10 and 3600) into its memory. Since the data written in the internal memory remains stored despite a power cut, each time the sensor is powered up again, it starts measuring

automatically at the last period written in the memory. That way, `getRHT()` and `getCO2P()` functions do not measure the air but read the last values in the internal memory. However, if the user wishes to measure at a precise moment by himself, he can request the sensor sampling by reading the status byte or executing the `trigger_measurement()` function. If the last measurement taken by the CO₂ sensor dates back for more than 10 seconds, then the sensor will take a new measurement. However, if the last measurement is more recent than 10 seconds, the sensor will not take a new one. In order to be sure that the sensor performs a new measurement, regardless of the time interval that has elapsed, the following procedure should be followed: request a new measurement, wait 10 seconds, and trigger another measurement. In this way, regardless of whether the last measurement is old enough or not, the conditions will be fulfilled for the second triggering to occur because there will always be a minimum of 10 seconds elapsed since the last measurement. This procedure is automatically performed by function `trigger_measurement(force=True)`. After each triggered measurement, the sensor interval time counter is set back to zero, and the following measurement will therefore occur when the sensor sampling period is elapsed.

Figure 6 is the sensor timing schematic, as given by the manufacturer. Note that the sensor's firmware manages the blue part on the graph while the central computer controls the red/green part. Sampling initiation (Power-Up) occurs autonomously at regular intervals as defined in the sensor memory under the sampling period or manually when triggered by the user, as explained before. Contrary to what one might think by looking at the diagram, the sensor does not notify the central computer when the measurement is complete. It is, therefore, impossible to know when the data are ready for reading unless manual triggering and close monitoring of the time elapsed via the software. The software should wait a minimum of 15 seconds between measurement triggering and data reading (recommended by the manufacturer).

In order to reduce that sampling uncertainty, the Seacairy software has been designed to proceed as follows. At startup, the central computer writes into the sensor memory the sampling period. Then, it requests a measurement via the `trigger_measurement()` function. Ten seconds later, the Seacairy software loop starts. At the start of each new loop, the central

computer sends a measurement request to the sensor. Then, it waits for a minimum of 18 seconds⁵ before reading the data into the CO₂ sensor memory.

| | Minimum | Typical | Maximum |
|--|---------|---------|---------|
| t_{pwrap} * (power up) | 4.7s | | 16.2s |
| t_{meas} (measurement) | | 0.8s | |
| t_{mti} (measurement time interval) $\pm 6.25\%$ | 15s | | 3600s |

* see chapter 6.1

Examples:

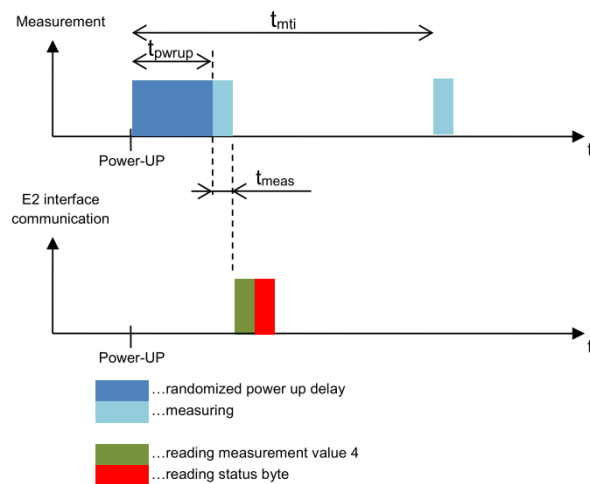


Figure 6 CO₂ sensor sampling timing

Source: manufacturer's sensor documentation [8]

1.6 Faced issues during the development

1.6.1 Consecutive I²C write and read

In contrast with the standard I²C protocol (refer to point 1.1 on page 5), the CO₂ communication works by combining write (from Master to Slave) and read (from Slave to Master) actions in one single sentence, without any STOP bit in between, as shown in Figure 7. Where most devices using I²C communication separate write and read operations into two separate sentences, the CO₂ sensor does not. After intensive research and trials, it appears that the standard SMBUS Python library usually used for I²C communications had been updated for such devices under the name `smbus2` [18]. Figure 7 is the schematic transmission to be followed by the central computer to read relative humidity and temperature and Figure 8 is a Python code template for doing such I²C operation. Note that the green parts in the schematic go from the CO₂ sensor to the central computer and the white parts from the central computer to the sensor. The ACK (acknowledge) bits indicate successful reception of the last transmitted byte, while a

⁵ Other tasks are processed by the central computer during this delay. If those task takes extra time, then the delay may increase.

NACK (non-acknowledge) indicates an error when receiving the previous byte, a complete memory, or the end of a read operation. CS means clock stretching, the situation where the slave keeps the SCK low to make the master wait.

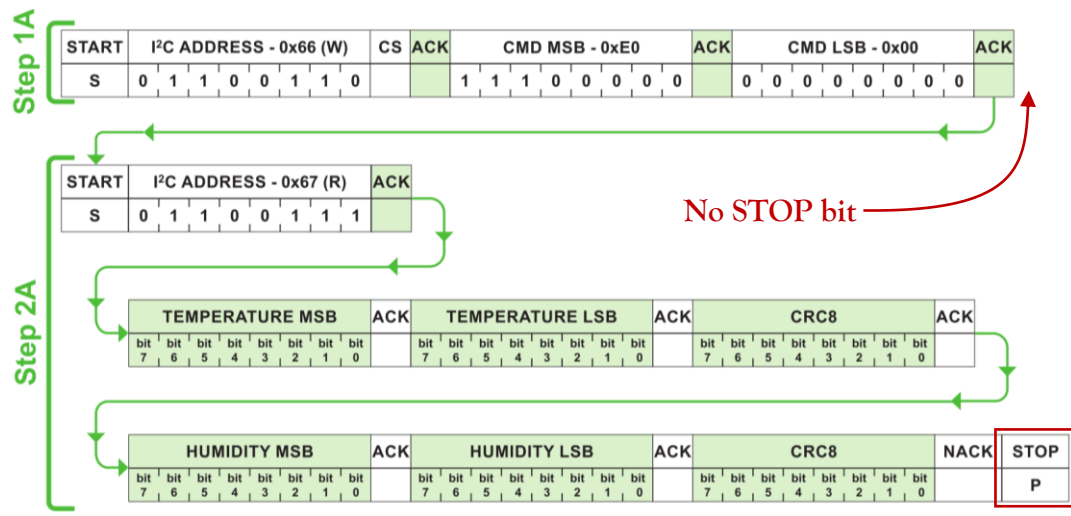


Figure 7 Schematic of I²C write and read sentences without any STOP bit in between

Source: E+E Elektronik documentation [9]

```
from smbus2 import SMBus, i2c_msg # import libraries after pip3 install
smbus2 (in the virtual environment)

CO2_address = 0x33 # CO2 sensor i2c slave address

# Indicate bytes to be written to the CO2 sensor
# (See sensor documentation to know which bytes to send for which purpose)
write = i2c_msg.write(CO2_address, [0xE0, 0x00])

# Indicate amount of bytes to be read
read = i2c_msg.read(CO2_address, 6)

with SMBus(1) as bus:
    # Start the communication, and indicate the i2c sentence between
    brackets
    # START + SLAVE ADDRESS & write bit + [master send bytes] + SLAVE
    ADDRESS & read bit + [slave send bytes] + STOP

    bus.i2c_rdwr(write, read)

    # Bytes read from the i2c communication are returned under the 'read'
    variable
    print(read) # will print on the screen the bytes sent by the sensor in
    a list
```

Figure 8 I²C write and read sentences without any STOP bit in between (Python code)

Source: adapted from Python Package Index (PyPi) [18]

1.6.2 Addition of other I²C devices to the central computer

As explained in point 1.1, all the devices using the I²C communication are connected to the central computer in parallel, some having as standard some pull-up resistors on SDA and

SCK lines welded on their PCBs. For the CO₂ sensor, the user must add these pull-up resistors manually when wiring the sensor. However, some devices such as the Sensirion Mass Flow Meter and the RTC already have their own pull-up resistances on both lines. Therefore, the more devices are connected to the central computer in parallel (as this is the way of connecting I²C devices), the more resistors are introduced in parallel, and the more the equivalent resistance⁶ will decrease⁷. Once these equivalent resistors drop below **10 kΩ**, which is the minimum value recommended by the manufacturer, the CO₂ sensor might not work correctly. After connecting the Sensirion Mass Flow Meter to the central computer, the CO₂ sensor stopped working. By removing the two pull-up resistors from the CO₂ sensor and unwelding the resistors on the RTC chip, the CO₂ sensor started working correctly again.

1.6.3 inability to manually trigger a measurement

According to the manufacturer's documentation, reading the status byte 0x71 should trigger a new measurement if the previous one is older than 10 seconds (see 1.5) and return one byte indicating the last measurement status. In practice, reading the status byte never trigger any measurement. Experiments were conducted to assess the ability of the central computer to send a sampling request to the sensor. Firstly, the sensor measurement period was set to 60 seconds. Then, the software was adapted to read the status byte every 10 seconds. The experiment was carried out in the dark to see the sensor's infrared light during the air sampling. It appears that only measurements every 60 seconds have taken place. Therefore, this means that the sensor does not react to any sampling requests from the central computer. In conclusion, it is impossible to request a measurement at a specific desired time.

1.6.4 Continuous indication of temperature error on the status byte

As shown in Figure 6 on page 15, the manufacturer indicates that the central computer should read the status byte (0x71) after each reading of the sensor measurements to check the status of the last measurement taken. When converting the value read into a binary, eight digits are obtained, either 0 or 1. Bits 0, 1 and 3⁸ indicate humidity, temperature, and the CO₂ sensor status, respectively. A value of 1 on these bits indicates a problem, while a value of 0 shows none.

⁶ The equivalent resistance is the resistance that could replace all other resistance in the electronic circuit without changing the conductivity properties of that circuit.

⁷ By increasing the resistance, the circuit tends to be open, while by decreasing the resistance it tends to be closed (Ohm's law).

⁸ Note that the bits are read from right to left, and bit 0 is the first on the right.

Each time the status has been read by the central computer, especially when triggering a measurement, the CO₂ sensor returned an error for the temperature sensor. The only plausible reason is a too low electrical voltage. However, the power supply is constant at 5 volts.

1.6.5 Checksum error during measurement readings

The Python software redundantly returns errors and warning on the screen during the temperature, relative humidity, pressure, and CO₂ measurements readings. In practice, as shown in Figure 4 on page 13, the sensor sends a series of bytes containing the data and the result of a known calculation made based on the bytes previously transmitted (this value is called the checksum). So, after reading all the data bytes from the sensor, the central computer performs the same calculation. Therefore, it should find the same answer, the opposite showing that some data bytes were not correctly received. While the Seacanairy is running, the checksum Python algorithm, performing that verification, often detects a mismatch between the two checksums and prints an error message on the screen. Another message follows, showing the value of the data bytes, the sensor checksum and the central computer calculated checksum. When taking a closer look at the information, we notice that similar values keep coming back. This suggests that they might be error codes, but the manufacturer's documentation never talks about it. When such errors occur, the software waits for 3 seconds and start rereading the measurements data. The Python software repeated this read and wait for the loop a certain number of times until the maximal reading attempt number is reached. This value can be changed in the `seacanairy_settings.yaml` file, the default value being six. The value 6 comes from several tests conducted during the development of the Seacanairy. On the one hand, this leaves enough chance for the sensor to return the data correctly. On the other hand, this prevents the software loop from taking too much time.

2 OPC-N3 particulate matter sensor

The OPC-N3 is a sensor manufactured by Alphasense designed to measure particulate matter. As its name suggests, this sensor is an Optical Particulate Counter. A fan forces air to move through the sensor measuring chamber. Then, a laser beam illuminates the air, the beam being so thin that it allows the illumination of only one aerosol at a time. When the laser beam hits an aerosol, the laser light is scattered by the particle. The intensity of the light reflected makes it possible to determine the type, and the mass of the particle illuminated [14,32]. The

sensor can detect around 100% of particles with a size of $0.35\ \mu\text{m}$ and around 50% of particles of $0.30\ \mu\text{m}$ diameter. The sensor measuring range (from 0.35 to $40\ \mu\text{m}$) is divided into 24 bins. For each bin size, the Alphasense proprietary software counts the number of particulates passed during one second. The firmware is fast enough to reach 10,000 particle readings per second. Then, according to the particle size distribution, the sensor returns three mass loadings, respectively PM_{10} (total mass of particles smaller than one-micrometre particles), $\text{PM}_{2.5}$ (total mass of particles smaller than $2.5\ \mu\text{m}$) and $\text{PM}_{1.0}$ (total mass of 10-micrometre particles). Switching automatically to low and high gain can manage reading PM_{10} of up to $10,000\ \mu\text{g}/\text{m}^3$ [2]. The primary sensor characteristics are summarized in Table 5. After emails exchange with the sensor manufacturer, it appears that the total flow rate is the total amount of air that passes through the sensor, propelled by the fan, while the sample flow rate is the air flow that passes through the laser beam.

Table 5 OPC-N3 sensor characteristics

Source: adapted from the official documentation [23]

| | | |
|--|--|----------------------|
| Particulate size range | 0.35 → 40 μm spherical equivalent size (based on 100% detection efficiency at $0.35\ \mu\text{m}$, 50% at $0.3\ \mu\text{m}$) | |
| Total flow rate ⁹ (typical) | 5.5 L/min | |
| Sample flow rate ¹⁰ (typical) | 280 mL/min | |
| Max particle count rate | 10 000 particles/second | |
| Max coincidence probability | At 10^6 particles/L | 0.84 % concentration |
| | At 500 particles/L | 0.24 % concentration |



Figure 9 Pictures of the OPC-N3

Source: Alphasense website [24]

⁹ Total amount of air that passes through the sensor, propelled by the fan.

¹⁰ Air flow that passes through the laser beam.

2.1 Data returned by the sensor

During each sampling, the sensor returns a histogram containing a whole range of data. Hereafter a list of all the information given by the sensor, based on the manufacturer's documentation and mail exchanged with the sensor designers [2,3].

- Bin from 0 to 23: number of particles for each bin size passed through the laser beam per minute.
- MToF (mean time of flight): for bin 1, bin 3, bin 5 and bin 7, it is the average amount of time that particles (for corresponding bin size) took to cross the laser beam. The sensor uses these values for dynamic fan compensation.
- Sampling period: the amount of time the laser beam has been analyzing the air. This value is always half the time when both fan and laser run because the sensor automatically samples in low and high gain.
- Sample flow rate: air flow rate passing through the laser beam. This value is always lower than the total air flow rate because the laser only measures a part of the total amount of air passing through the sampling space.
- Temperature: this value should not be considered because the temperature sensor is not located in the fan air flow. With the temperature sensor being welded to the OPC-N3's motherboard, temperature readings are always higher than other more exact temperature sensors (the CO₂ sensor in the case of the Seacanairy).
- Relative humidity: as for the temperature reading, this measurement should not be considered.
- PM₁: the total mass of particles smaller than one-micrometre particles.
- PM_{2.5}: the total mass of particles smaller than 2.5-micrometre particule.
- PM₁₀: the total mass of 10-micrometre particles.
- Reject count Glitch: noise and invalid particle errors indication.

- Reject count Long TOF: number of particles rejected by the system because of their too long flight time in the laser beam.
- Reject count ratio.
- Fan rev count. Value has always been zero.
- Laser status.

2.2 Sensor communication and wiring

As shown in Figure 9, the sensor has two connexions. The first one is the Serial Peripheral Interface (SPI) available through a Molex Pico-Clasp PCB Header fitted with a single Row, six pins and a pitch socket of 1 mm. Table 6 shows the necessary components that should be purchased to make the connection to the SPI possible. The second connection is a USB micro-B. Through those two sockets, the sensor can be operated in three ways.

Table 6 Compatible sockets with the OPC-N3

Source: manufacturer's documentation [2], own work

| No. | Reference | Name | Price per unit |
|-----|-------------|---|----------------|
| 1 | 15133-0603 | Cable Pico-Clasp to Pico-Clasp assembly 1 row, 6 way, 300 mm length | 6.6€ |
| 2 | 501331-0607 | Male PCB Molex connector | 6.61/10 units |

The first operating way is the standalone mode. Once the sensor is powered by 5 Volts via pin numbers 1 and 6 (see Annexe 3 on page 117), and if any communications occur for one minute on the SPI lines, the sensor starts sampling by itself. Both fan and laser keep running, and the sensor firmware periodically stores data in the built-in SD card, which is then accessible via the USB port. However, this autonomous working system is not suitable within the Seacairy framework. The second way of employing this sensor is by using the Alphasense OPC-N3 software with an SPI to USB adapter. Running on Windows, it allows easy use of the sensor and quick display of the measurements. Nevertheless, this does not meet the requirements of the Seacairy. The last operating principle relies on custom software. After reading the manufacturer's documentation and hours of testing, it was possible to create our own Python software allowing the excellent operation of the OPC-N3 from the central computer.

The Serial Peripheral Interface is a simultaneous synchronized transmission where both the master and the slave communicate at the same time bit per bit based on the master's clock pulse. Therefore, as shown in Figure 10, the protocol requires minimum of three wires: the MOSI (Master Out Slave In), the MISO (Master In Slave Out) and the SCLK (Serial Clock) controlled by the master. Annexe 3 on page 117 shows the wiring of the OPC-N3 sensor to the central computer. It gets power via pins 1 and 5 from the Raspberry Pi. Then, pins 2, 3 and 4 (SCLK, MISO and MOSI) must be connected to the Raspberry Pi GPIO on the correct pins [3]. The SPI protocol allows several SPI devices to connect in parallel to the same bus (SCLK, MISO and MOSI), provided they all carry a SS (Slave Select), also called CS (chip select) line. In this case, the Master will keep the SS line of the sensor with which it wishes to communicate low, indicating to the other sensors whose SS line is high that they must remain silent. The OPC-N3 slave select line is available through pin 5. This line should be connected to the central computer SPI CE0 line (Raspberry Pi HAT pin 26). Nevertheless, the use of the SS line with the OPC-N3 has never worked correctly during the Seacanairy development. After personal investigations, it appears that the SS line of the OPC-N3 must remain connected to the ground. This is explained in deeper detail in point 2.5.2 on page 34. The Seacanairy software operates the OPC-N3 communications based on the Linux Kernel Python library named `spidev` [34].

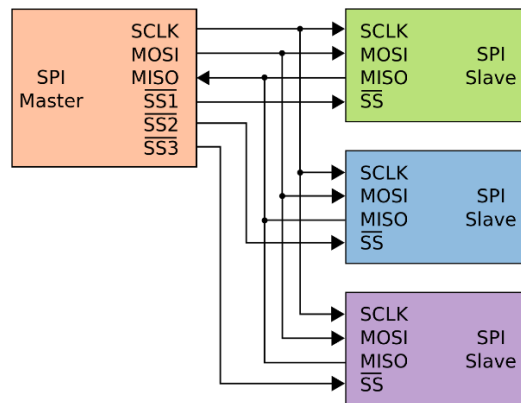


Figure 10 Schématic of the connection of multiple SPI devices to the same Master

Source: Wikipedia, Serial Peripheral Interface [4]

2.3 Software function list

Python software has been written based on the manufacturer's official documentation and Python libraries documentation to control the sensor from the central computer. Developing the software required hours of research, either in examples found on the internet or trial and error. In the end, it handles all the communications, the data verification, the

conversion of the transmitted bytes into measurements, and the modification of parameters within the sensor itself. Table 5 list all the functions of the OPC-N3 Python software. Note that `get_data()` is the final function that does all the necessary operations to get a measurement (start the fan, the laser, sample, bytes download and conversion, and stop fan and laser). A copy of the Python file (`OPCN3.py`) is available in Annexe 6 on page 134.

Table 7 OPCN3.py list of functions

Source own work, with the help of the manufacturer's documentation [2,3]

| Function | Goal | Argument | Return |
|---|---|---|---|
| <code>initiate_transmission</code> (<code>command_byte</code>) ¹¹ | Initiate SPI transmission to the OPC-N3 First loop on the manufacturer's flow Chart | <code>command_byte</code> : byte to be sent during communication initiation | True when SPI initiation has been done, False if it failed |
| <code>fan_off()</code> | Turn OFF the fan of the OPC-N3 | None | False if it succeeded turning off the fan, True if it failed |
| <code>fan_on()</code> | Turn ON the fan of the OPC-N3 | None | True if it succeeded in turning off the fan, False if it failed |
| <code>laser_on()</code> | Turn ON the laser of the OPC-N3 | None | True if it succeeded in turning off the laser, False if it failed |
| <code>laser_off()</code> | Turn OFF the laser of the OPC-N3 | None | False if it succeeded turning off the laser, True if it failed |
| <code>read_DAC_power_status</code> (<code>item='all'</code>) | Read the status of the Digital to Analog Converter as well as the Power Status Try only one time to read the byte(s) | <code>item</code> : 'fan', 'laser', 'fanDAC', 'laserDAC', 'laser_switch', 'gain', 'auto_gain_toggle', 'all' | DAC power byte, 5 status bytes if argument is 'all' |
| <code>digest(data)</code> | Calculate the CRC8 Checksum with the bytes received | <code>data</code> : a list containing an infinite number of bytes with which to calculate the checksum | Calculated checksum |

¹¹ Refer to Figure 11 on page 43 for a schematic function flowchart, and point 2.4.1 on page 41 for its explanation.

| Function | Goal | Argument | Return |
|---------------------------------------|---|---|--|
| <code>check(checksum, *data)</code> | Check that the data received are correct, based on those data and the checksum given | <code>checksum</code> : checksum sent by the sensor (the last byte in any transmission) <code>data</code> : bytes sent by the sensor, with which to calculate the checksum | True if data are corrects, False if they are not |
| <code>convert_IEEE754(value)</code> | Join bytes and convert them to float according to the IEEE754 encryption | <code>value</code> : a list containing the two bytes to decrypt | decrypted float |
| <code>loading_bar(name, delay)</code> | Show a loading bar on the screen for a certain amount of time. Make the user understand the software is doing/waiting for something | <code>name</code> : text to be shown on the left of the loading bar (waiting, sampling...) <code>delay</code> : the amount of time the system is waiting (seconds) | Nothing |
| <code>PM_reading()</code> | Read the PM bytes only from the OPC-N3 sensor Read the data and convert them in a readable format, checksum enabled Does neither start the fan nor start the laser Recommended to use <code>read_histogram()</code> instead of this function | None | List[PM 1, PM2.5, PM10] |

| Function | Goal | Argument | Return |
|---|--|--|--|
| <p style="text-align: center;">getPM (flushing_time, sampling_time)</p> | <p>Get PM measurement from OPC-N3 Recommended to use get_data() instead of this function</p> | <p>flushing_time: time (seconds) during which the fan runs alone to flush the sensor with fresh air sampling_time: time (seconds) during which the laser reads the particulate matter in the air</p> | <p>List[PM1, PM2.5, PM10]</p> |
| <p style="text-align: center;">read_histogram (sampling_period)¹²</p> | <p>Read all the available data from the OPC-N3 It first read the histogram to remove the old data remaining in the OPCN3 buffer Then it lets the sensor take sample during the defined sampling period Finally, it read a last time the histogram data returned by the sensor It decodes the bytes returned into a readable format It returns everything in a dictionary</p> | <p>sampling_period: the amount of time (seconds) during while the fan is running and the laser is sampling</p> | <p>Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative humidity", "bin", "MToF", "sampling time", "sample flow rate", "reject count glitch", "reject count longTOF", "reject count ratio", "reject count out of range", "fan revolution count", "laser status"}</p> |

¹² Refer to Figure 12 on page 44 for schematic function flowchart, and point 2.4.2 on page 41 for its explanation.

| Function | Goal | Argument | Return |
|--|--|---|---|
| get_data (flushing_time, sampling_time) | Get all the possible data from the OPCN3 sensor Start the fan, flush air during defined time, start the laser, sample the air during defined time, turn off the laser and the fan | flushing_time: time during which the ventilator is running without sampling to refresh the air inside the casing sampling_time: time during which the sensor is sampling | Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative humidity", "bin", "MToF", "sampling time", "sample flow rate", "reject count glitch", "reject count longTOF", "reject count ratio", "reject count out of range", "fan revolution count", "laser status"} |
| join_bytes (list_of_bytes) | Join bytes to an integer, from byte 0 to byte infinite (right to the left) | list_of_bytes: list [bytes coming from the spi.readbytes or spi.xfer functions] | Bytes concatenated to an integer |
| set_fan_speed (speed_percent) | Set the sensor fan speed Reduce fan speed can decrease dust deposition in the sensor casing The argument in percent, calibrated from the slowest as possible to the fastest | speed_percent: number between 0 and 100 (0 = slowest, 100 = fastest) | Nothing |
| initialization_SPI () | Initialize the OPCN3 SPI To be executed once after Seacanairy power up To be executed on time only after powering up the OPCN3 | Nothing | Nothing |

2.4 Software schematic

The flowcharts on the following pages have been drawn up based on the Python software written previously to allow the correct integration of the OPC-N3 into the Seacanairy framework. After long hours of working, Python software meets the Seacanairy requirements and operates the OPC-N3 properly. The purpose of the following flowcharts is to illustrate the different interactions and processes that occur during the execution of certain functions of the software.

2.4.1 SPI communication initiation

Each communication with the OPC-N3 begins with an initiation loop. First, the central computer sends a byte on the SPI (called the command byte) to indicate the start of a certain type of communication to the sensor. Then, the sensor sends back to the central computer whether it is ready to carry out the asked communication. If the sensor returns that it is busy (0x31), then the central computer waits ten milliseconds and tries again. This call operation can be repeated up to 60 times. If even after 60 calls, the sensor is still not ready, then the central computer Python software considers a communication misunderstanding. Therefore, it waits for 3 seconds, the amount of time required by the sensor to clear its SPI cache after the communication issue. This is represented by 'cycle' in the diagram. When these two loops have taken place three times in a row, then the Seacanairy software considers the error to be more severe and cancels the initial operation. This procedure is shown in Figure 11.

2.4.2 Histogram reading

Figure 12 on page 32 shows all the steps performed by the software to get the histogram data. In the diagram, the transmission initiation is performed by the `initiate_transmission(command_byte)` function, as explained in point 2.4.1 and shown in Figure 11. If this initiation process fails, then the histogram reading is cancelled. In order to receive data from the OPC-N3, the central computer must send bytes on the SPI to the sensor and simultaneously read its answer (the value of the bytes sent does not matter).

The histogram consists of 85 bytes containing all the data. The 86th byte sent results from a known calculation made by the sensor with the previous 85 bytes, which is the checksum. When the central computer has finished reading the 86 bytes, it performs the same calculation with the 85 bytes transmitted and then compares its answer with the sensor checksum. If the two values are the same, then it means that the 85 bytes received are correct. However, if the two

values are different, then it means that one or more data bytes were corrupted during the SPI transmission. In such a situation, a second reading is therefore necessary. However, each time the central computer reads the 86 bytes, the OPC-N3 deletes all the measurement related data and starts counting particles from zero. This also explains why the 86 bytes of the histogram are read at the start of the histogram reading procedure. However, it also means that if the checksums are different, the central computer has to wait again for the OPC-N3 to measure the air before reading a second time. This is the reason why there is a loop in between the second initiate transmission process and the checksum decision diamond. Note that this feature can be disabled via the settings file (see page 98).

2.4.3 *Perform a particulate matter measurement*

Figure 13 indicates the different operations carried out by the `get_data()` function in order to obtain a histogram from the OPC-N3. If the ventilator fails to start, then the software will neither start the laser nor read the histogram. Indeed, no measurement is possible without any air flow in the sampling chamber. Likewise, the central computer will not read the histogram data if it failed to start the laser.

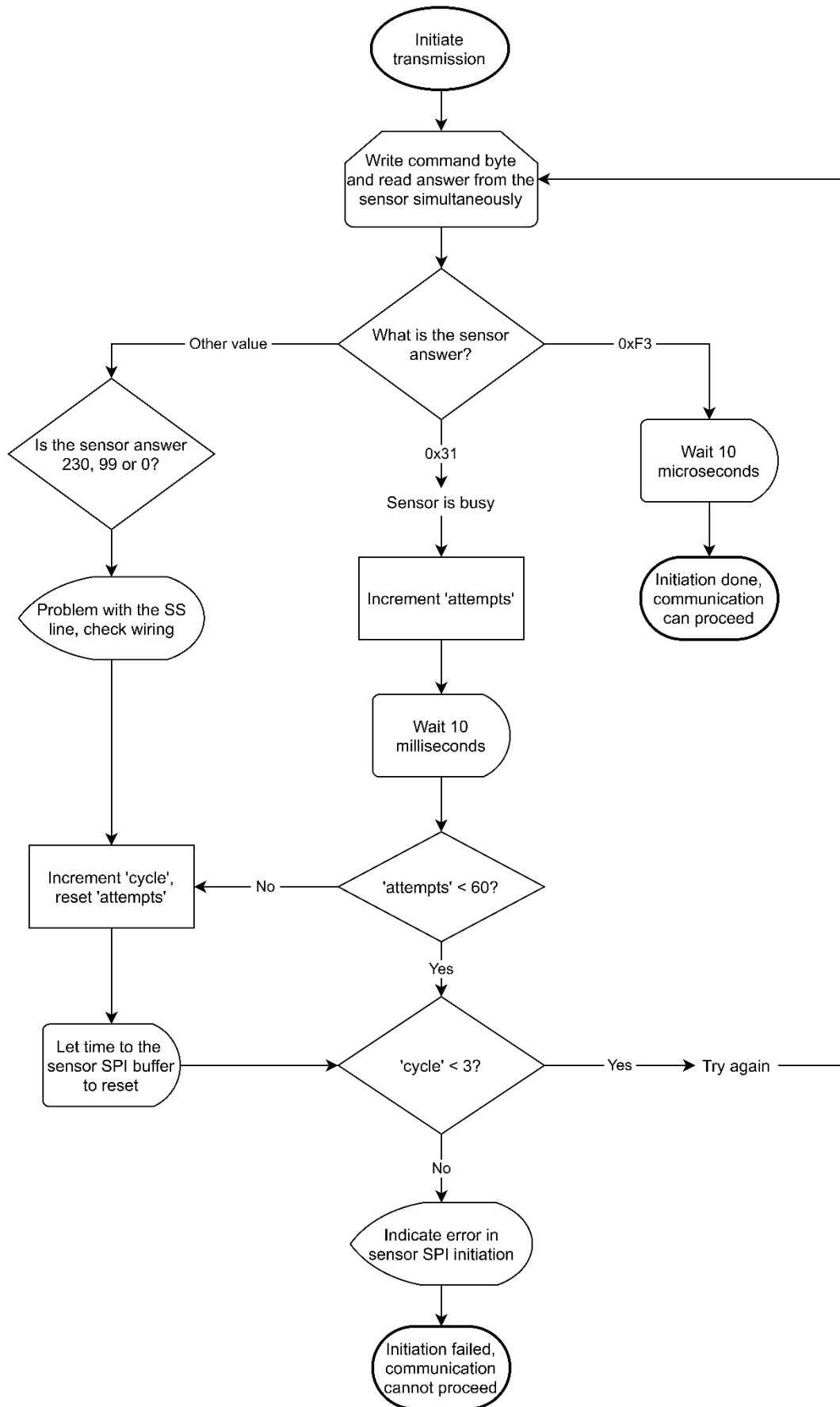


Figure 11 Flowchart of the SPI communication initiation

Source: own work, based on the manufacturer's documentation [2,3]

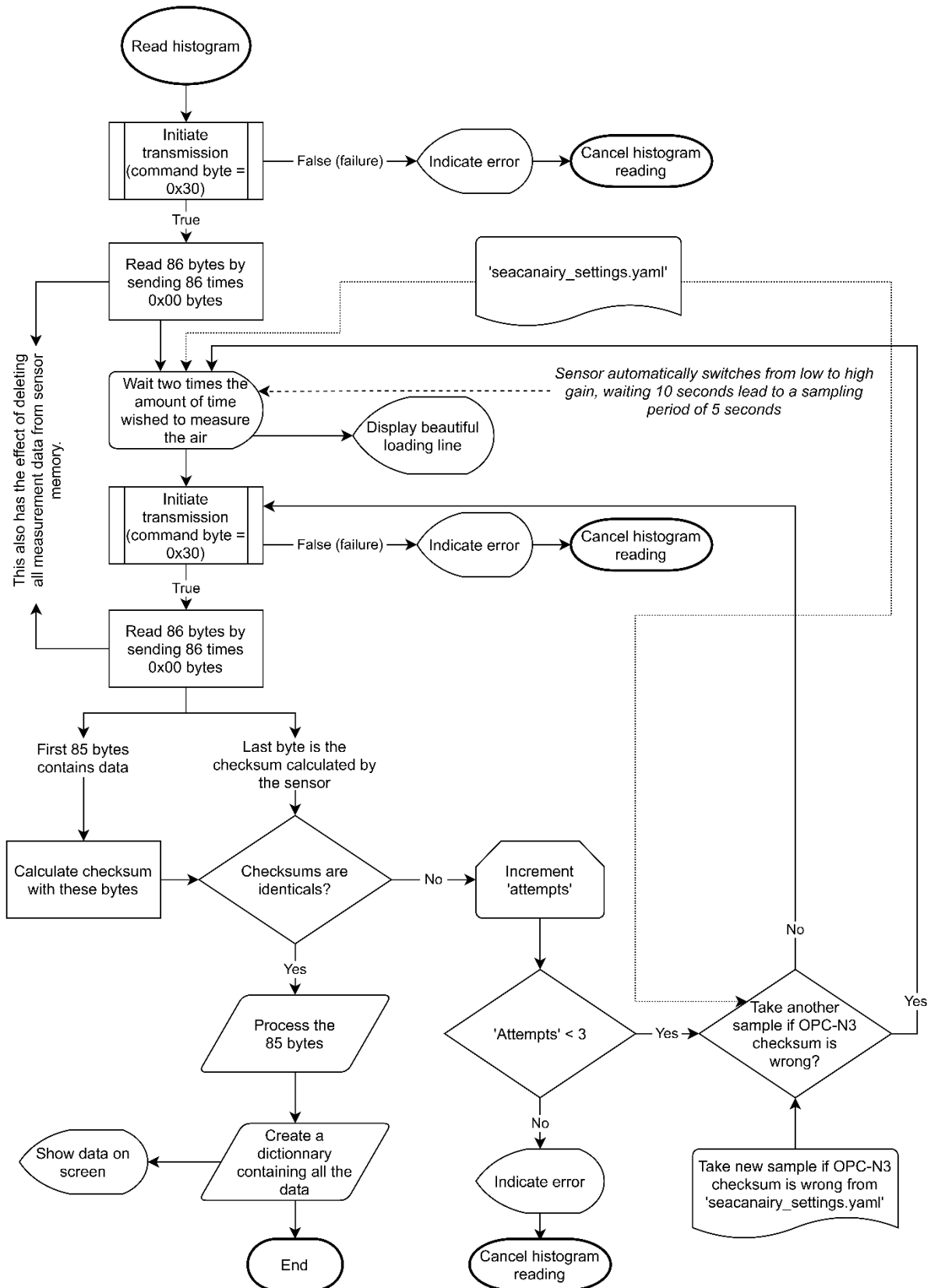


Figure 12 Flowchart of the histogram reading

Source: own work, based on the manufacturer's documentation [2,3]

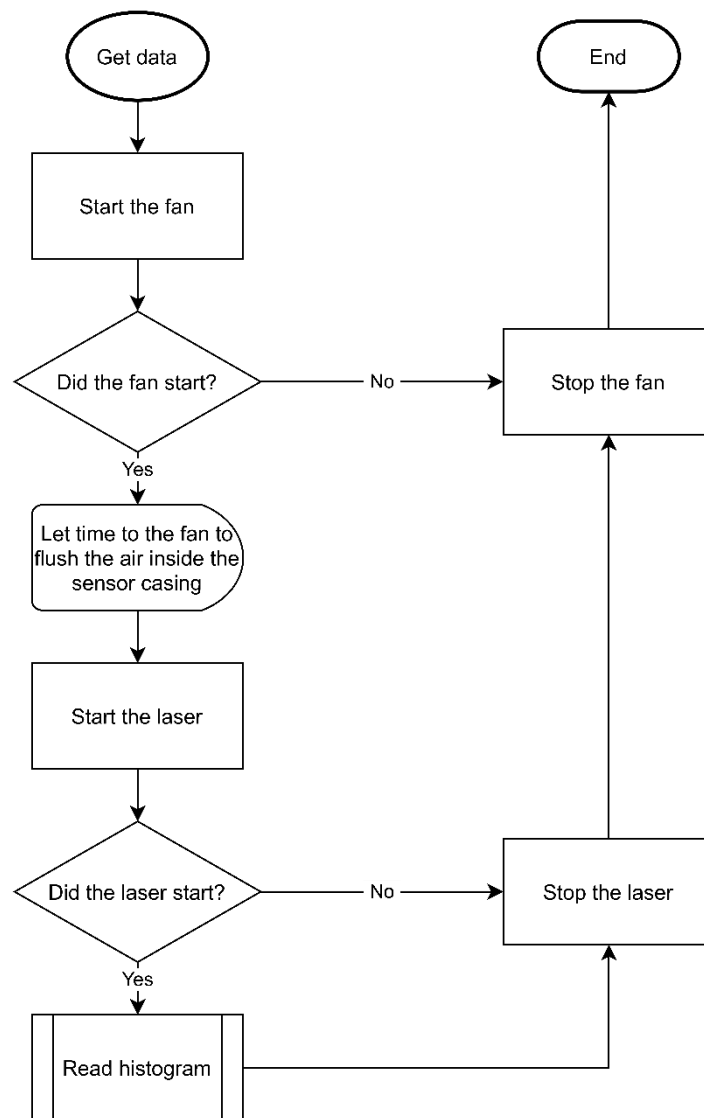


Figure 13 Flowchart of getting data from the OPC-N3 sensor

Source: own work, based on the sensor manufacturer's documentation [2,3]

2.5 Faced issues

2.5.1 Simultaneous reading and writing of data

In opposition to the I²C communication explained previously for the CO₂ sensor, Serial Protocol Interface and the OPC-N3 use two-way simultaneous communication. This means that the central computer sends bytes at the same time as others are received from the sensor. Therefore, writing data with function `spi.writebytes()` and then reading a certain number of bytes with function `spi.readbytes()` does not work because it is imperative to perform these two actions simultaneously. The only function allowing such operation is `spi.xfer()`, which writes the bytes inserted as arguments and returns the bytes received simultaneously. Note that those functions are provided by the `spidev` Python library [34].

2.5.2 Sensor Slave Select line wiring

Pin 5 in Annexe 3 on page 117 is the SS (Slave Select) line. When low, it indicates that the current communication on the SPI bus is assigned to the OPC-N3. On the other hand, when this line is high, it tells the sensor to remain silent on the SPI bus. As our central computer Python library (`spidev`) is compatible with the use of such a line on GPIO pin 24 (see Annexe 3 on page 117), the OPC-N3 SS line was initially connected through this pin to the central computer. However, during software development, while initiating SPI communication (as explained in 2.4.1 on page 29), the OPC-N3 never responded bytes `0xF3` or `0x31` as expected, but instead the following combinations: `[230, 99, 0]`, or `[36, 146, 73]`. Long research followed to find out if the error came from the Raspberry Pi (the central computer used), the Python library that providing the SPI communications, the wiring of the sensor, or the sensor itself. When moving to the SS line wiring, it was noticed that the values received during SPI communication initiation varied. From then on, research focused on the SS line. After many tests, the addition of resistors, and even diodes, it was discovered that the sensor worked perfectly and reliably when the SS line remains continuously connected to the ground. Therefore, this discovery goes against the manufacturer's documentation, which explicitly indicates that the SS line can be left disconnected [3].

2.5.3 Frequency conflict between the OPC-N3 SPI and the GPS UART

While developing and connecting the GPS receiver (see point 4 on page 45) to the UART GPIO bus of the central computer, all communication with the OPC-N3 on the SPI bus were lost. Despite the excellent functioning of the OPC-N3 initiation process, the reading of the 86 histogram bytes gave alternately 0 and 46. Several tests were achieved out. Firstly, the SPI bus of the Raspberry Pi was tested by connecting both the MISO and the MOSI of the Raspberry Pi through a $10\text{ k}\Omega$ resistor¹³. Then, we wrote a small Python software which had the simple purpose of sending a series of bytes¹⁴ (on the MOSI) while simultaneously reading the received bytes (on the MISO). This test was successful because all the numbers sent were correctly received by the Raspberry Pi. So, this means that the Raspberry Pi SPI bus was working correctly. Secondly, several SPI frequencies were tested, from 300 to 750 kHz (speed range defined by the manufacturer), but this did not solve the problem [3]. After reflection, the idea arises that the frequencies of the two buses (UART of the GPS and SPI of OPC-N3) could conflict. Let imagine

¹³ The resistor allows any damage to the Raspberry Pi due to the short circuit thus formed.

¹⁴ A simple `spi.xfer([0, 1, 2, 3...])` from the `spidev` Python library.

that one bus is operating at a particular frequency while another is operating on another. The CPU must remain stable on a fixed frequency during the transmission to detect bus rising and lowering for the communication to work. If the SPI and UART frequencies are not multiple of each other, then the processor may miss bits on one of these buses. As the UART frequency is fixed at 9600 by the GPS receiver clock, adaptations should be applied to the SPI communication frequency of the OPC-N3. The following formula shows the calculation performed in order to find the used frequency of **307200 Hz**. At this frequency, the OPC-N3 usually worked again.

$$9600 \times 31 = 297600 \rightarrow \text{outside of OPC-N3 working frequency}$$

$$9600 \times 32 = 307200 \rightarrow \text{inside OPC-N3 working frequency}$$

2.6 Interference between M&C air pump and SPI communication

When adding the M&C air pump to the installation, all communication with the OPC-N3 were lost. Two hypotheses were exposed: the physical vibrations of the suitcase and the electrical noise caused by the pump motor. Hereafter is a summary of the tests performed:

- The OPC-N3 has been removed from the case and hold in hands to be isolated from any physical vibration generated by the pump motor. No improvements were noticed.
- Le top aluminium plate also suffers from vibrations. The central computer, fixed on that plate, was detached and held in hands to remain isolated from any physical vibration. No improvements were noticed.
- The cable provided is formed of six individuals strands. All strands were scotched together using insulation tape to avoid any cable movement. No improvements were observed.
- Several power supplies have been tested. Unfortunately, neither the Raspberry Pi nor the Traco Power 5V 4A power supply helps. Increasing/reducing tension does not change anything.
- The USB female socket on the OPC-N3 was used as an additional power supply. An old USB cable was cut to connect only the 5V and the ground cable. Increasing power supply capacity did not solve the issue.

- The power source of the air pump motor was separated from the sensor's power supply using two separate 220V cables and plugs. The electric isolation of the sensors and the motor did not resolve the problem.
- Increasing/decreasing SPI frequency does not increase communication efficiency. Transmitted bytes remains corrupted, and communication keeps failing.
- After disabling the air pump in the software, the OPC-N3 works back again.
- If the air pump never stops running, then the OPC-N3 works without any problem.
- The addition of an electric noise filter before the air pump motor improves the communication efficiency with the sensor.
- The addition of a delay between the start of the pump and the first communication increases the communications' efficiency.
- Earthing the aluminium plates do not improve the SPI efficiency.
- Removing the Traco power earth connection does not improve the SPI communications.
- Removing the pump motor earth connection does not improve the SPI communications.
- Increasing the 5V cable diameter between the power supply and the central computer should help to keep a stable 5V. However, no considerable improvements could be noticed.
- A capacitor of 1 Farad (5V) was connected in parallel to the power supply. This should help the power supply to keep a stable 5V output.

2.6.1 Isolation of the pump from the 220V line via a noise reducer

Adding a noise filter upstream of the pump power supply has reduced the number of OPC-N3 errors. The addition of this component is explained in more detail in point 0 on page 69.

2.6.2 Increasing the power supply capacity

Any DC voltage source (in this case, 5V) has a specific maximum power corresponding to its maximum current and voltage ($P = U \times I$). Modern voltage sources have an active regulation system that aims to actively rectify the current to a stable 5V regardless of the current delivered. However, when the maximum power is reached, the active regulation has no longer enough power in reserve to keep a stable voltage. Therefore, noise may appear at the output of the DC source, coming from either the 220V line or the rectifier. Therefore, it is recommended to install an oversized power supply to stay in the linear zone, the area where the system can reach the most stable DC output. For that reason, the voltage source supplied with the Raspberry Pi (central computer computing unit) has been replaced by another more powerful unit (4A instead of 2.5). As shown in Figure 14, two cables come out of the voltage source: one goes to the Raspberry Pi, the other a terminal block situated on the printed circuit board. In practice, the two are already connected via the 40 pins female header of the Raspberry Pi. However, the thin tracks of the printed circuit may generate some resistance, leading to a loss of voltage across the central computer [26].

In addition, a 1 Farad capacitor has been added at the output of the voltage source. Connected in parallel to the 5V DC lines, one loaded, it helps keeping a stable tension.

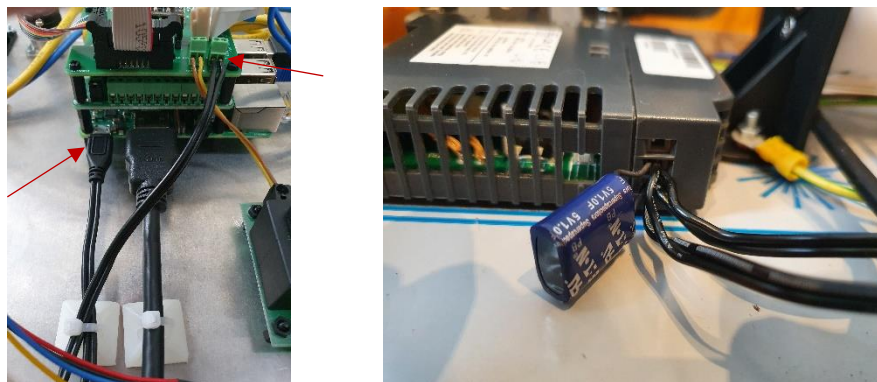


Figure 14 Power supply of the central computer (Raspberry Pi and printed circuit) and 1 Farad capacitance

Source: own work

2.6.3 *Addition of a rest period between starting the pump and the first communication with the sensor*

Inrush current, also known as Switch-on surge, is the maximum instantaneous current of an electric motor at the connection to a voltage source. The sudden increase in current generates a deformation of the 220V sinusoid, leading to waves, resonances and harmonics oscillating for a certain amount of time in the 220V lines. Those waves can disturb the 5V DC voltage source, leading to noise propagation to the central computer. Then, the noise spreads to the SPI clock line. Peaks and valleys on the clock line disrupt the OPC-N3, synchronization between the central computer and the sensor is lost, and bits are corrupted. This assumption stands because adding a delay between the start of the electric motor and the first communication helps troubleshoot the communication issues. Delay should be between 5 and 10 seconds.

3 The 4-AFE gas sensors board from Alphasense

The 4-AFE is a gas sensor holder produced by Alphasense, allowing the easy connection of four gas sensors and one temperature sensor. The set allows the measurement of nitrogen dioxide (NO₂), ozone (O₃), sulphur dioxide (SO₂) as well as carbon monoxide (CO). The sensor is supplied with 5V by a linear voltage power supply to reduce the noise in the measurements as much as possible. Each sensor returns two analogue voltages (main and auxiliary voltages), which the central computer's ADC converts into numerical values. Then, the voltages measured can be converted into gas concentrations through calibration. Table 8 shows all the products required for using this sensor.

Table 8 Inventory of the Alphasense gas sensor

Source: own work, adapted from the Master thesis of Lukas Van der Borghet [37]

| No. | Piece No. | Description | Price per unit |
|-----|----------------|--|----------------|
| 1 | NO2-A43F | Alphasense Nitrogen Dioxide electrochemical sensor | 48.00 |
| 2 | OX-A431 | Alphasense ozone electrochemical sensor | 50.00 |
| 3 | SO2-A4 | Alphasense sulphur dioxide electrochemical sensor | 48.00 |
| 4 | CO-A4 | Alphasense carbon monoxide electrochemical sensor | 48.00 |
| 5 | 810-0023-00 | Alphasense 4-AFE board | 152.00 |
| 6 | 000-CBLE-03 | Alphasense 4-AFE Board cable board-to-board | 10.00 |
| 7 | PSU30205 | Lascar 240V/5V 100 mA linear power supply | 40.00 |
| 8 | PL-16ADC | Alchemy Power Analogue to Digital Converter for Raspberry Pi | 50.00 |
| 9 | C05a-12-ASB1-G | Valcon Wire-to-Board 2mm Straight PCB IDC Latched Headers | ~ 0.50 |

3.1 Wiring of the 4-AFE board and the Analog to Digital Converter (ADC)

Figure 15 shows the wiring of the central computer sensor. Note that the 5V Lascar linear power supply powers all the gas sensors while the ADC is powered by 5V from the Raspberry Pi via the HAT sockets. For the ADC to measure the voltage on the pins coming from the gas sensors, the ADC, the sensors and the power supply must share the same ground. Therefore, all the GNDs in the diagram are connected together [37]. Annexe 3 on page 117 is a schematic representation of the wiring of the different sensors from the Alphasense sensor board to the ADC via the PCB.

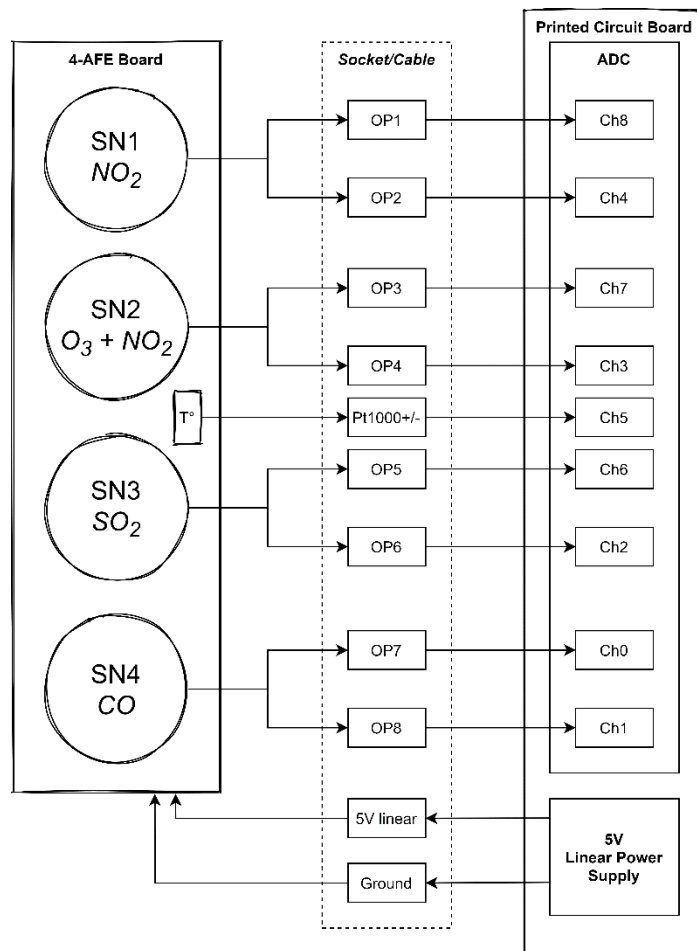


Figure 15 Schematic representation of the Alphasense 4-AFE wiring

Source: own work, using draw.io

3.2 Software function list

A Python code has been written to measure the electrical voltages from gas sensors via the Central computer's Analog to Digital Converter (ADC). The software code (`AFE.py`, available in Annexe 7 on page 154) is an improved copy of the code written by Lukas Van der Borghet for his Master thesis [37]. Table 7 is a list of all the software functions.

Table 9 OPCN3.py list of functions

Source own work, some part of code coming from Lukas Van der Borgh't's thesis master [37]

| Function | Goal | Argument | Return |
|--|--|--|---|
| getADCreading (adc_address, adc_channel) | Read tension from the ADC on a certain channel | adc_address: slave i2c address adc_channel: channel where to read tension | Tension between channel and ground (volts) |
| get_temp() | Measure tension of the temperature sensor (Note that the sensor is not located in the gas hood.) | None | Dictionary containing tension in milli volts {'temperature raw'} |
| get_NO2() | Measure tension of NO2 main and auxiliary electrodes | None | Dictionary containing tensions in milli volts {'NO2 main', 'NO2 aux'} |
| get_OX() | Measure tension of OX main and auxiliary electrodes | None | Dictionary containing tensions in milli volts {'OX main', 'OX aux'} |
| get_SO2() | Measure tension of SO2 main and auxiliary electrodes | None | Dictionary containing tensions in milli volts {'SO2 main', 'SO2 aux'} |
| get_CO() | Measure tension of CO main and auxiliary electrodes | None | Dictionary containing tensions in milli volts {'CO main', 'CO aux'} |

| Function | Goal | Argument | Return |
|---|--|---|--|
| apply_calibration(dictionary) | Apply calibration to the tensions measured before [Not yet finished] | dictionary: dictionary containing tensions from other functions | Initial dictionary with ppm concentration added {'NO2 ppm', 'OX ppm', 'SO2 ppm', 'CO ppm', 'temperature'} |
| get_data() | Get all available data from the 4-AFE Alphasense Board (one single instantaneous reading) | None | Dictionary{'NO2 ppm', 'NO2 main', 'NO2 aux', 'OX ppm', 'OX main', 'OX aux', 'SO2 ppm', 'SO2 main', 'SO2 aux', 'CO ppm', 'CO main', 'CO aux', 'temperature', 'temperature raw'} |
| start_averaged_data (number_of_measurements) | Perform multiple readings and makes an average Run <code>get_averaged_data()</code> once thread is finished to get the data Improved for threading application (no display prints) | number_of_measurements: number of measurement to average, each single measurement taking around 2 seconds | Dictionary{'NO2 main', 'NO2 aux', 'OX main', 'OX aux', 'SO2 main', 'SO2 aux', 'CO main', 'CO aux', 'temperature raw'} |

| Function | Goal | Argument | Return |
|--|--|--|----------------|
| <pre> start_background_ average_measurement (number_of_measurements, delay=0) </pre> | <p>Start a new thread to perform averaged reading in the background</p> <p>Run <code>get_averaged_data()</code> once the thread is finished to get the data</p> <pre> thread = threading.Thread(target=AFE.start_averaged_data, args=(number_of_measurements, delay), daemon=True) </pre> <p>in your own code is preferred</p> | <pre> number_of_measurements: number of measurements to average, each single measurement taking around 2 seconds delay: amount of time in between the start of the thread and the start of the sampling operation </pre> | <p>Nothing</p> |

| Function | Goal | Argument | Return |
|---|--|-------------|---|
| <p style="text-align: center;"><code>get_averaged_data()</code></p> | <p>Read the data of the last <code>start_averaged_data()</code> (or <code>start_background_average_measurement()</code> performed</p> | <p>None</p> | <p>dictionary{'NO2 ppm', 'NO2 main', 'NO2 aux', 'NO2 main max', 'NO2 main min', 'NO2 aux max', 'NO2 aux min', 'OX ppm', 'OX main', 'OX aux', 'OX main max', 'OX main min', 'OX aux max', 'OX aux min', 'SO2 ppm', 'SO2 main', 'SO2 aux', 'SO2 main max', 'SO2 main min', 'SO2 aux max', 'SO2 aux min', 'CO ppm', 'CO main', 'CO aux', 'CO main max', 'CO main min', 'CO aux max', 'CO aux min', 'temperature', 'temperature raw'}</p> |

3.3 Analogic signal noise reduction

Previous research performed by Lukas Van der Borgh on the 4-AFE board shows that there is some noise on the analogue signal [37]. In order to reduce this noise, a function has been added to the Python software. Function `start_averaged_data(number_of_measurements)` takes several successive measurements of the five sensors and then calculated the average. The function also returns the minimum and maximum value read to get an idea of the margin of error. As this operation takes around two seconds for each reading loop, the function has been written so that it can run silent in the background in a separated Python thread, allowing the central computer to perform other tasks simultaneously. A second function is necessary to retrieve measurements taken in the background. In conclusion, `start_averaged_data(number_of_measurements)` takes multiple measurements and calculates an average, while `get_averaged_data()` returns the results of the last averaged measurement. Note that the number of successive measurements taken can be adapted in the Seacanairy settings file (see point 3 on page 98).

3.4 Calibration

The calibration from Lukas Van der Borgh has been added to the software [37]. That way, the systems performs the necessary calculations to convert the tensions from the gas sensors in millivolts to concentrations in ppb. Files containing the calibration settings is shown in Figure 72 on page 109, and an example of this file is available in Annexe 12 on page 192.

4 The GPS receiver

It has been shown that the speed of a ship has a considerable influence on air pollution measurements taken on board. As part of the design of an instrument for measuring air pollution optimized for use on a ship, it is interesting to relate gas and particulate matter measurements with ship speed and position simultaneously. For this purpose, a GPS received has been connected to the central computer. It consists of the sensor board VMA430 manufactured by Velleman in which the receiver U-BLOX NEO-7M is incorporated.

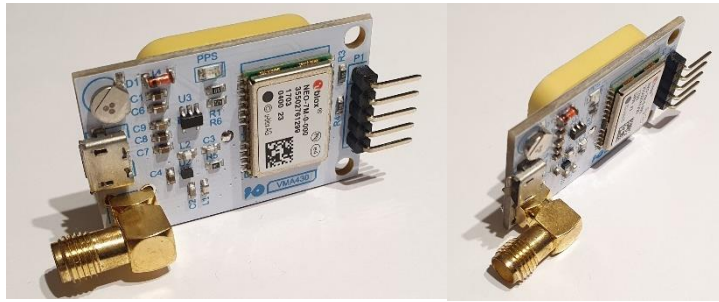


Figure 16 Velleman VMA430 and U-BLOX NEO-7M chip

Source: own pictures

4.1 Wiring of the GPS receiver

Unlike the other sensors explained previously in this paper, the GPS receiver uses UART (Universal Asynchronous Receiver Transmitter) communication. This communication protocol is made possible by two wires only: the TX (the line on which the data goes out) and the RX (the line on which the data enters). Unlike the SPI and I²C ports, there is no clock synchronizing communications, and there is neither Master nor Slave. Therefore, both devices must transfer their data at the same speed, 9600 baud rate, in the case of the GPS receiver. Annexe 3 on page 117 shows the wiring from the GPS receiver to the host computer. The RX pins are connected to the TXs, and the TXs are connected to the RXs. The PPS (time pulse) is the pulse at which the GPS receiver sends location data. This line is not necessary.

4.2 Software function list

The process for obtaining the position data from the GPS is much more straightforward than the other sensors explained previously. Therefore, there is only one important function. Table 10 indicates the function and what it returns. A copy of the code is available in Annexe 8 on page 168. Note that the software written could also work with other GPS receivers than the one used as long as their data are structured in the same way (NMEA 180¹⁵) and work with UART.

Table 10 GPS.py function list

Source: own work

¹⁵ Standard for communicating position, heading, speed and time data via several lines of text, whose numerical data is separated by commas.

| Function | Goal | Argument | Return |
|-----------------------------|--|----------|--|
| <code>get_position()</code> | Read position data from the GPS receiver | None | Dictionary{fix time, fix date, fix date and time, latitude, longitude, SOG, COG, status, horizontal precision, altitude, WGS84 correction, current time, accuracy, fix status} |

4.3 Faced issues

4.3.1 Frequency conflict between the OPC-N3 SPI and the GPS UART

The addition of the GPS receiver and the activation of the UART port interfered with the proper functioning of the OPC-N3. After some extensive research, the hypothesis emerged of a conflict between the frequencies of the SPI port of the OPC-N3 and the UART port of the GPS receiver. This problem and its solution are explained in point 2.5.3 on page 34.

4.3.2 Random UART port opening problem

After resolving the conflict issue between the SPI port and UART port, another issue appeared. Randomly, the Python software fails to activate the UART port. Since it cannot be started, it is therefore impossible to read the data from the GPS receiver.

The Raspberry Pi 3B + has two UART ports. The first one is the PL011. This port is controlled by an independent chip and is therefore neither influenced by the CPU workload nor its frequency. The PL011 is, therefore, more stable, more reliable, and performs better in the background. The second UART port is called the miniUART, a virtual port directly connected to the CPU, accessible from the Raspberry Pi purchase via the GPIOs (pin 8 and 10). Therefore, a variation in the frequency of the processor has a direct influence on the miniUART behaviour. Remember that the communication protocol is asynchronous and that there is no clock line between the devices connected to that bus. Devices are therefore supposed to communicate at a constant and stable frequency. Therefore, a slight variation in the CPU frequency can disrupt the miniUART and consequently induce corruption or loss of data. Initially, the Raspberry Pi's Bluetooth is connected to the PL011, so Raspberry Pi users are supposed to use the miniUART for their electronic projects. However, when the Raspberry Pi workload increases (internet, TeamViewer, the addition of sensors activation of multiple communication protocols), the miniUART is no longer accurate enough. Therefore, it is necessary to apply changes in the Raspberry Pi operating system to deactivate the Bluetooth and divert the PL011 on the GPIO pins [21,22,28,35].

Before making any changes to the Raspberry Pi file system, it is better to back up the whole SD card, using Win32 Disk Imager. In this way, if we were to miss a step and the Raspberry Pi no longer starts, we can put the backup back on the SD card and start over on the old configuration.

- Start by executing the display command that lists the Raspberry Pi's various ports: `ls -l /dev*`. In the list, if `serial0` fit with `tty0`, then it means that Bluetooth is connected to PL011. Then the further steps must be applied. If `serial0` go with `ttyAMA0`, then it means that PL011 is already connected to the GPIO and that no further steps are necessary [21].

```

brw-rw---- 1 root disk    1,  1 avr  6 15:19 ram1
brw-rw---- 1 root disk    1, 10 avr  6 15:19 ram10
brw-rw---- 1 root disk    1, 11 avr  6 15:19 ram11
brw-rw---- 1 root disk    1, 12 avr  6 15:19 ram12
brw-rw---- 1 root disk    1, 13 avr  6 15:19 ram13
brw-rw---- 1 root disk    1, 14 avr  6 15:19 ram14
brw-rw---- 1 root disk    1, 15 avr  6 15:19 ram15
brw-rw---- 1 root disk    1,  2 avr  6 15:19 ram2
brw-rw---- 1 root disk    1,  3 avr  6 15:19 ram3
brw-rw---- 1 root disk    1,  4 avr  6 15:19 ram4
brw-rw---- 1 root disk    1,  5 avr  6 15:19 ram5
brw-rw---- 1 root disk    1,  6 avr  6 15:19 ram6
brw-rw---- 1 root disk    1,  7 avr  6 15:19 ram7
brw-rw---- 1 root disk    1,  8 avr  6 15:19 ram8
brw-rw---- 1 root disk    1,  9 avr  6 15:19 ram9
crw-rw-rw- 1 root root    1,  8 avr  6 15:19 random
drwxr-xr-x 2 root root    60 jan  1 1970 raw
crw-rw-r-- 1 root netdev 10, 242 avr  6 15:19 rfkill
lrwxrwxrwx 1 root root     5 avr  6 15:19 serial0 -> ttyS0
lrwxrwxrwx 1 root root     7 avr  6 15:19 serial1 -> ttyAMA0
drwxrwxrwt 2 root root    40 fév 14 2019 shm
drwxr-xr-x 3 root root   180 avr  6 15:19 snd
crw-rw---- 1 root spi    153,  0 avr  6 15:19 spidev0.0
crw-rw---- 1 root spi    153,  1 avr  6 15:19 spidev0.1
lrwxrwxrwx 1 root root    15 fév 14 2019 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root    15 fév 14 2019 stdin  -> /proc/self/fd/0
lrwxrwxrwx 1 root root    15 fév 14 2019 stdout -> /proc/self/fd/1
crw-rw-rw- 1 root tty     5,  0 avr  6 15:19 tty
crw-rw---- 1 root tty     4,  0 avr  6 15:19 tty0
crw----- 1 pi  tty     4,  1 avr  6 15:19 tty1
crw-rw---- 1 root tty     4, 10 avr  6 15:19 tty10
crw-rw---- 1 root tty     4, 11 avr  6 15:19 tty11

```

Figure 17 `ls -l /dev*` on Raspberry Pi, UART configuration

Source: own work

- Open a new terminal and type the function: `sudo nano /boot/config.txt`. After execution, a nano GNU will open. Scroll to the bottom of the file and add `dtoverlay=pi3-disable_bt`. The addition of comments as done in Figure 18 helps keeping a track in the system modifications. Finally, to save and exit the nano GNU, press on `ctrl+x`, then press `enter` and `o` (do not confuse the letter with the number zero) [21].

```

GNU nano 3.2 config.txt
# NOOBS Auto-generated Settings:
# 1-wire settings
dtoverlay=w1-gpio
max_usb_current=1
hdmi_group=2
hdmi_mode=87
hdmi_cvt 1024 600 60 6 0 0 0
hdmi_drive=1

enable_uart=1

# Added by Cyril Dewez on the 06-04-21
# Disable the Bluetooth to recover the good UART port for GPS
dtoverlay=pi3-disable-bt

```

Figure 18 Switching UARTs on the Raspberry Pi (config.txt)

Source: own work

- Open a new terminal and execute the following function:
`sudo nano /boot/cmdline.txt`. Remove from the file the following:
`console=serial0,115200`. Close nano GNU by pressing `ctrl+x`, then `enter` and `o` (do not confuse the letter with the number zero) [21].
- Disable the Bluetooth UART service by executing the following command:
`sudo systemctl disable hciuart`. A message should then indicate the service deactivation [21].
- Restart the Raspberry Pi. After boot, check that both UARTs has been well swiped. Open a new terminal and execute `ls -l /dev*`. Now, `serial0` should be `ttyAMA0`. If it is not the case, then reboot again [21].

5 Sensirion Mass Flow Meter

The flow sensor uses the I²C protocol as the CO₂ sensor. The technical specificities of the communication protocol have already been explained in point 1.1 on page 5. By measuring only one parameter, the air flow, the use of this sensor is much easier. No problem was encountered. The Python code for the use of this sensor can be found in Annexe 9 on page 176, and the diagram of the electrical connections is in Annexe 3 on page 117.

6 The RTC (real-time clock) - DS3231

When the central computer gets out of power, it lost track of time and restarts on January 1, 2000. Indeed, the Raspberry Pi has been designed to stay connected to the internet, where it synchronizes to server time over the cloud. However, for the Seacanairy to link the measurement with the time and date, a Real-Time Clock must be connected to the central computer. The module used is the DS3231. The sensor works via I²C communication. The RTC wiring is shown in Annexe 3 on page 117. Pin 1 fits with the pin at the bottom of Figure 19.



Figure 19 RTC DS3231 chip

Source: own picture

6.1 Faced issues

6.1.1 I²C pull-up resistors

When adding the RTC and the Sensirion Mass Flow Meter to the Seacanairy, the central computer lost communication with the CO₂ sensor. The cause is the excess of resistors connected in parallel to the I²C bus. As explained in point 1.6.2 on page 17, the RTC has its own welded pull-up resistors. However, these resistors are already introduced by another sensor connected to the same bus, and the resistances of the RTC are therefore in excess. Therefore, they should be removed. Notice that in Figure 19, two resistors have been unsoldered at one of their ends.

6.1.2 Integration on the PCB

During the design of the PCB (version 2.0), the pin order was reversed. It induced the connection of the RTC in the other direction, hindering the connection of the CO₂ sensor. To avoid ordering a new circuit board, the female connection of the RTC has been unsoldered and resoldered on the other side. That way, the connexion of both the RTC and the CO₂ sensor is possible. Figure 20 shows the normal shape on the left and the modification on the right. Version 3.0 of the PCB resolves this error (see Annexe 4 on page 119 for a schematic of the two different versions).

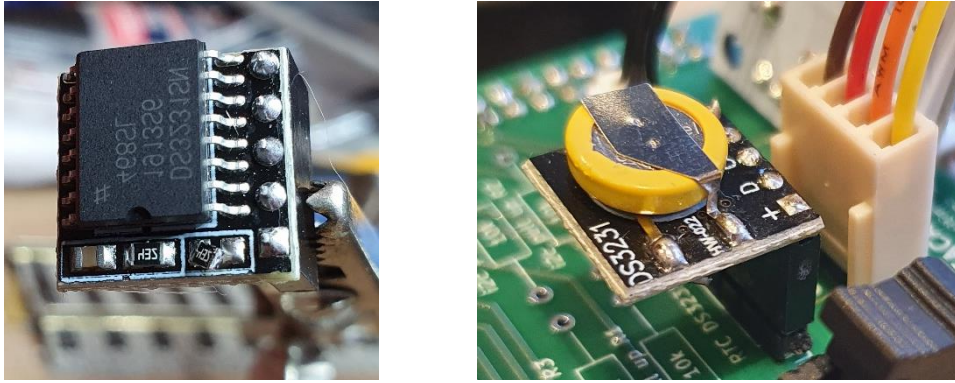


Figure 20 Relocation of the DS3231 socket to solve the PCB design problem

Source: own work

Chapter 2

Combining components into a measuring device

This chapter focuses on connecting all the components (explained in detail in the earlier chapter) to create one instrument. Sensors are interconnected in diverse ways. Firstly, a tube connects all the sensors in series, and a pump pushes a controlled amount of air through the sensors. Secondly, sensors are connected according to their specific wiring characteristics to the central computer via a custom motherboard. Finally, there is a physical connection between the sampling instrument and the suitcase.



Figure 21 The case of the Seacanairy

Source: own work

1 Building the device into a transportable suitcase

The goal of the Seacanairy was to develop a measuring device inside a suitcase so that it is portable and can be used in maritime conditions. This means that the measuring system must be incorporated into an impact resistant box that is watertight and easily transportable. For this, a yellow-coloured Pelican Storm Case iM2720 was selected. The case has an O-ring sealing that protects the instruments from water jets, and the plastic is impact resistant. It is large enough (55.9 cm x 43.2 cm x 25.4 cm) for future improvements. Instruments are fixed on three aluminium plates installed in the casing. Table I shows the required components.

Table 11 Inventory of the components needed to build the casing

Source: own work

| No. | Supplier | Piece No. | Name | Price per unit | Quantity | Price |
|-----|-----------------------|------------------|---|---|------------------|------------|
| 1 | Pelican | iM2720 | Pelican Storm Case, yellow, 55.9 x 43.2 x 25.4 cm | 306.50 | 1 | 306.50 |
| 2 | | iM27XX- BEZEL | Bezel-Kit Lid iM27XX for Peli Storm Case iM2700 iM2720 iM2750 <i>The frame on which the lid and top plates are placed.</i> | 95.78 | 2 | 191.56 |
| 3 | John Steel | | Raw and shiny aluminium, filmed, 2 mm thickness <i>Case Panels¹⁶</i> | | 3 | 106.08 |
| 4 | Clabots ¹⁷ | M5 | Screws to fix the frames to the case | | 8 | Negligible |
| | | | Autoblocking nuts | | 8 | |
| 5 | | M4 | Screws to fix the top and lid plates to their frames | | 16 ¹⁸ | |
| 6 | | M6 | Screws to fix the bottom plate to the case | | 4 | |
| | | | Autoblocking nuts | | 8 | |
| | | | Washers <i>Placed inside the case to divide the load from the bold on the plastic</i> | | 4 | |
| 7 | | | | Watertight gaskets <i>Tighten screws fixing bottom plate, placed outside of the case</i> | | |

¹⁶ See Annexe 2 on page 85 for their schematics.¹⁷ Local do-it-yourself supplier.¹⁸ The number of screws also depends on the accuracy of the drillings position.

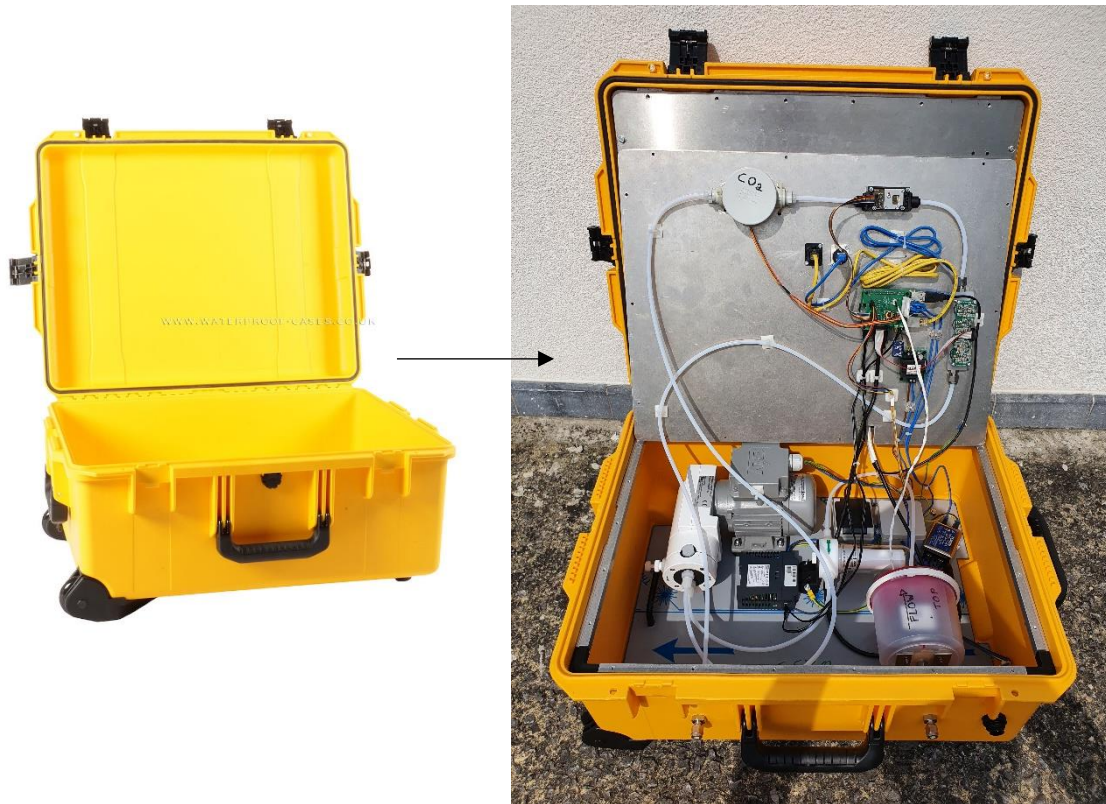


Figure 22 Pelican Storm Case iM2720 before/after
 Source: *Waterproof Cases [25] (left), own work (right)*

1.1 Three aluminium plates in the casing

Three aluminium plates must be fixed in the casing using frames and screws to install all the components in the case. In that way, the case contains three levels of aluminium plates.

1.2 The bottom plate

The bottom plate is fixed on four M6 bolts fixed through the bottom of the case. Bolts are fitted with anti-vibration screws and watertight washers (see Figure 23 on page 57). The dimensions of the plate can be found in Figure 74 on page 116. The following components are fixed to the bottom plate: the M&C air pump, the 220V electricity junction box, a 220V plug, the 220V noise filter, the particulate matter sensor (OPC-N3), and the air filter (see Figure 24 on page 57). Components have been disposed to keep some free space for further improvements. Also, the layout must be thought out so that no component touches each other when closing.



Figure 23 Bottom plate fixing bolts (and the four bolts)

Source: own work

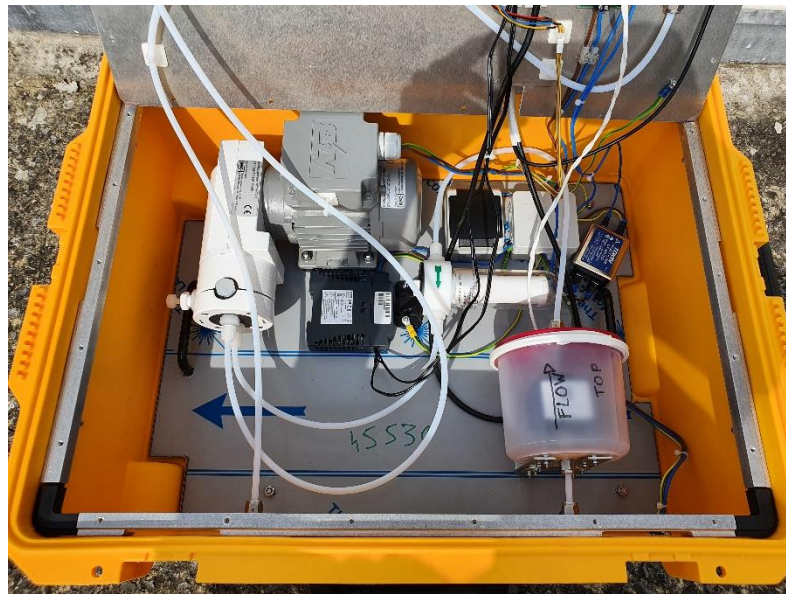


Figure 24 Picture of the bottom plate and its components

Source: own work

1.3 The cover plate in the case lid

The cover plate is placed on a frame via four M4 bolts (see point 1.5 on page 58 concerning drilling), and the frame is fixed into the lid with twenty rivets. All holes made inside the case are waterproof via rubbered washers included in the bezel kit's purchased package. Cover plates hold the touchscreen and the GPS receiver (see Figure 25). A cut-out on the hinge side plate allows cables to pass from the lower part to the upper part. Dimensions of the aluminium plate can be found in Annexe 2 on page 115.



Figure 25 Cover plate (in the case lid), back and front side

Source: own work

1.4 The top plate

The top plate is placed on a frame via four M4 bolts (see point 1.5 concerning drilling), and the frame is fixed to the case via eight M5 bolts. That way, the frame can be detached if necessary. All holes made inside the case are waterproof via rubbered washers included in the bezel kit's purchased package. Dimensions of the aluminium plate can be found in Annexe 2 on page 115. The top frame contains on the front a USB and ethernet plug connected to the central computer. The backside contains all the sensors and electronical components, such as the central computer, the Linear 5V transformer, the CO₂ sensor, the Alphasense 4-AFE Gas sensor, and the Sensirion Mass Flow Meter. A cut-out on the hinge side plate allows cables to pass from the lower to the upper.

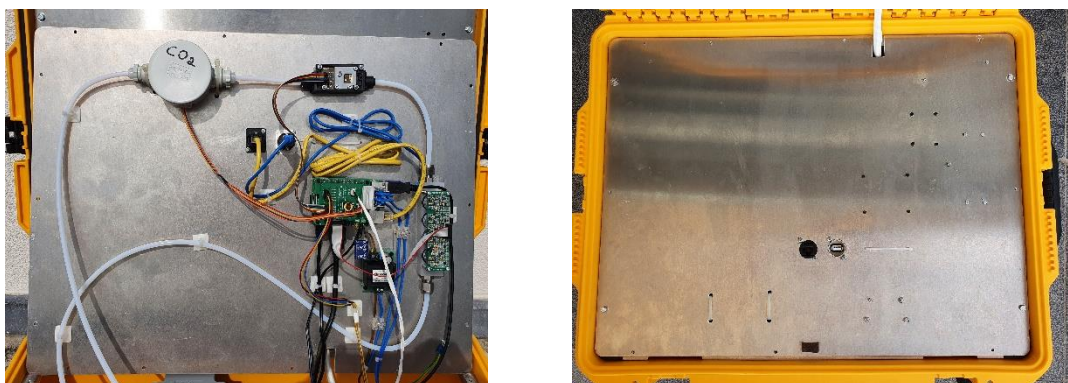


Figure 26 Top plate, back and front side

Source: own work

1.5 Drilling the plate to fix it on the frame

Since the suitcase is not perfectly rectangular, the size of the frame must be able to adapt to match the shape of the suitcase according to the desired height. This flexibility is achieved via the plastic sliding corners. Before drilling the holes in the plate, you must first position the frame

at the desired height, slide the corners so that the frame fits the shape of the suitcase, then remove the frame without changing the frame dimensions, and then drill the holes in the right places. Improper execution of this step leads to a deformation of the frame when tightening the frame fixing bolts and an offset between the holes in the frame and the holes drilled.

2 Connecting all sensors with tubes

In order to protect the sensors from the external environment and enable them to be calibrated, the external air to be measured must be conveyed to the sensors by means of tubes and a pump. The air is therefore forced to pass in series from one sensor to another. Therefore, in the next step, it will be possible to connect the measuring instrument to calibration gas cylinders to calibrate everything.

The Seacanairy is crossed by a tube in which circulates the air to be measured. This tube connects all the sensors in series (see Figure 24 on page 57), meaning that the same air passes from one sensor to another. As the sensors are not designed to be connected to a tube, they have been positioned in small tight boxes connected to the tube. At each tube intersection, there is a connector that allows the tightest possible junction between the tube and the box. This allows a pump to be located at the end of the line and to generate an airflow in the piping system. Figure 28 is a picture of the piping system of the Seacanairy, and Table 12 indicates all the components necessary to build a similar piping system. The order in which the sensors, pumps and filters were connected was decided as a result of the following thought. The most sensitive sensor in this project is the OPC-N3 due to its laser and air illumination system (more detailed explanation on page 19 at point 2). Particulate matter can stick along the tube's walls and accumulates behind junctions and other bends [2]. Therefore, the sensor tube must be as short and straight as possible to minimise the tube-induced error. On the other hand, the box containing the sensor does not support pressure. Therefore, it should be located on the suction side of the pump, and the vacuum will keep the box well closed and tight. The M&C pump has some ability to suck and blow air. The OPC-N3 and the filter were placed before the pump for the reasons explained above. In order not to overload the pump on the suction side (before the pump), the remaining sensors were positioned on the pressure side (after the pump).

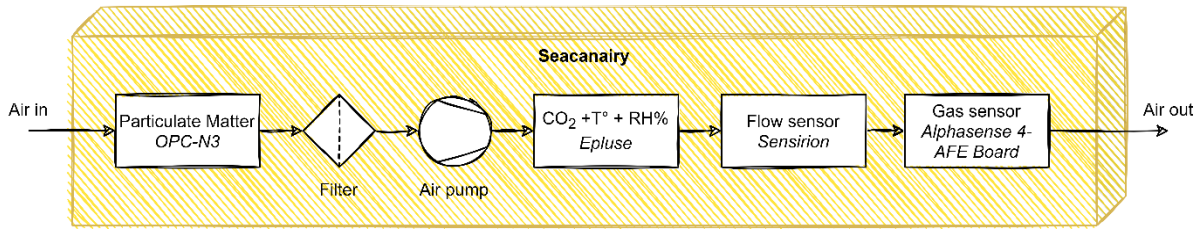


Figure 27 Schematic of the inboard piping system

Source: own work, using draw.io

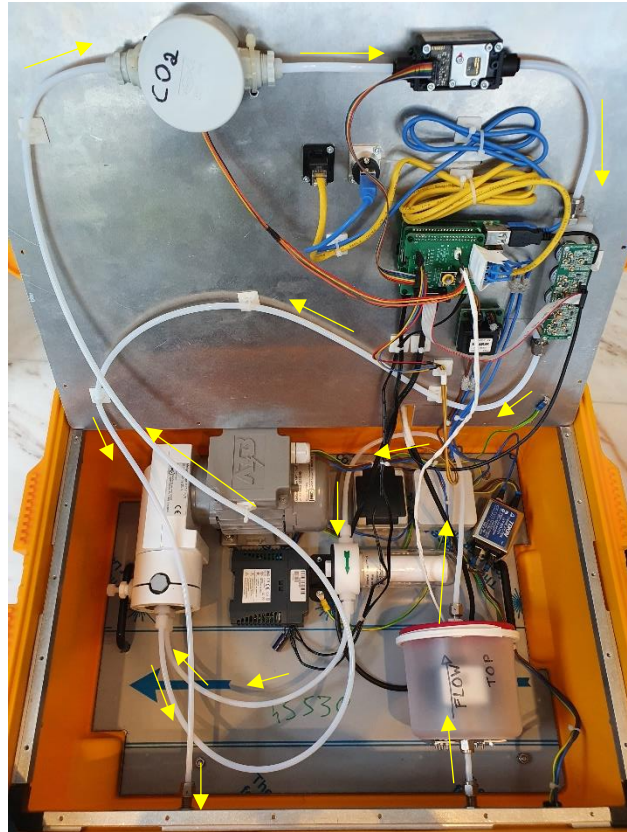


Figure 28 Picture of the piping system inside the Pelican case

Source: own work

Following goals were expected while designing the sampling system. Firstly, all the sensors must be kept protected from any physical impact and water splashes. This is guaranteed by the Pelican Case and watertight cable glands. Moreover, all the sensors must be connected to the same piping system. It is expected that they all measure the same air at the exact moment. Then, the piping system must be as gas-tight as possible to avoid any gas leakage and distortion in the measurements. Finally, the air volume inside the whole piping system must be as small as possible for better calibration. That way, the calibration process would require less calibration gas.

Table 12 Inventory of the piping system

Source: own work

| No. | Supplier | Piece No. | Name | Price per unit | Quantity | Price |
|-----|---------------------|-------------|---|----------------|--------------------|--------|
| 1 | M&C | 02B1000 | PTFE tube, DN 4/6 Sampling tube, internal diameter 4 mm, external diameter 6 mm | 10.20 | 10 m ¹⁹ | 102 |
| 2 | | 10T1000 | Hose cutter Hose cutter designed for precise perpendicular tube cuts | 22.70 | 1 | 22.70 |
| 3 | | 05V1060 | GE PVDF DN 4/6 - G 1/4" Connector from 4/6 tube to male screw type G size 1/4" Used for connecting the pump, the filter, and the CO ₂ sensor to the piping | 11.30 | 8 | 90.40 |
| 4 | | 05P1010 | MP-F 05 R, 230 V Bellow sampling pump, 320 NL/h, PTFE, needle valve | 971.00 | 1 | 971.00 |
| 5 | | 05P1050 | Mounting bracket for MP-F Support for the pump with anti-vibration pads | 59.00 | 1 | 59.00 |
| 6 | | 01F2200 | FT-2T Filter for universal gas sampling use | 490.00 | 1 | 490.00 |
| 7 | | 90F0002 | Filter element F-2T Compatible spare filter | 27.70 | 1 | 27,70 |
| 8 | Swagelok | SS-6M0-61 | Stainless Steel Swagelok Tube Fitting, Bulkhead Union, 6 mm Tube OD Case Inlet and Outlet connections, OPC-N3 box connectors | | 4 | |
| 9 | | SS-8M0-6-6M | Stainless Steel Swagelok Tube Fitting, Reducing Union, 8 mm x 6 mm Tube OD 8 mm outer diameter to 6 mm outer diameter tube converter Used in the OPC-N3 box | | 1 | |
| 10 | Brico ²⁰ | | Small cylindric electric junction box for the CO ₂ sensor box | | 1 | |
| 11 | | | Junction box electric connection gland and their screws | | 2 | |

¹⁹ This is the minimum length upon purchase.²⁰ Local do-it-yourself supplier.

| No. | Supplier | Piece No. | Name | Price per unit | Quantity | Price |
|-----|-----------------------|-----------|--|----------------|----------|-------|
| 12 | | | M3 threaded rods and their washers and bolts <i>For pulling the OPC-N3 sensor against a Swagelok connector and a gasket</i> | | 4 | |
| 13 | | | Teflon tape <i>To improve air system sealings</i> | 1.99 | 1 | 1.99 |
| 13 | Traffic ²⁰ | | Lunch box <i>OPC-N3 box</i> | 3.00 | 1 | 3.00 |

2.1.1 The use of PTFE tubes

An essential characteristic in the design is the tube material. Reactive gases can be absorbed by the piping material, the sensor housing, and the adapters, giving inaccurate low readings and an increased response time²¹. It is why some provisions should be taken while working with any piping or housing system [7].

Different materials are currently in use for gas sampling devices such as tubes, pumps, sensor housings, valves, and elbows. A popular material is PTFE (polytetrafluoroethylene), which belongs to the material type known as Teflon. The polymer is hydrophobic²² and has a low friction coefficient. PTFE is used in many applications that require high temperature, chemical resistance and low friction, such as wiring insulation, high-temperature protection, coating of non-stick pans, and lubrication [27]. PTFE mostly has a milky/white appearance. The second material often used in gas sampling applications is FEP (fluorinated ethylene propylene). FEP is chemically speaking similar to PTFE, except for its transparency and thermal resistance [12]. A last alternative material is Tygon (Saint-Gobain). It is a brand name to a large polymer tubing family. Some of them are composed of multiple layers of proprietary composition. Tygon is often used in the chemical industry, in laboratories and in pharmaceutical [29]. Some tubes are resistant to any chemical product, either gaseous, liquid or slurry [30].

In 2008, the Health and Safety Laboratory held a study for the Health and Safety Executive (English parliament) concerning the effect of tubing materials on gas detectors and sampling systems. Using gas detectors, different tubes materials, and tube diameters, they measured the delay between a change in gas concentration at the inlet of the tube and its

²¹ Amount of time between an event and its detection.

²² Which is not attracted by water.

detection at the end. They tested PTFE, FEP and Tygon tubes with Hydrogen Sulphide (H_2S), Nitrogen Dioxide (NO_2), Nitric Oxide (NO) and Toluene (C_7H_8). Their recommendations are summarized below [7].

- PTFE and FEP can be used with minimal effect when sampling H_2S , NO and NO_2 [7].
- Tygon may be used when sampling H_2S , NO and NO_2 if PTFE and FEP are not available providing the delay time is not an issue, but the tube dimensions must be as small as practically possible without restricting the flow rate of the sampling instrument [7].
- Tygon is not suitable for use in sampling C_7H_8 but PTFE and FEP may be used providing the delay time is not an issue [7].
- Taking into account the previous findings and the principal supplier catalogue (see Table 12 on page 61), it was decided to use PTFE pipings.

2.1.2 Air pump

As long as all the sensors are connected in sealed tubes, an air pump is necessary in order to circulate the outside air through all the collectors.

The air pump used (article 5 in Table 12 on page 61) was purchased at M&C. Its nominal flow rate is 5 slm (standard litre per minute²³), which corresponds to the air volume required by the OPC-N3 (particulate matter sensor). The 220V motor rotates a crankshaft converting the rotary motion into a reciprocating motion operating a PTFE bellow. For easier mounting inside the case, the pump head has been rotated²⁴ 90° to reduce its height. This pump contains only one bellow and is therefore single-acting: no air enters when air exits, and vice versa. As a result, the movement of the air is jerky, leading to some troubles for some sensors. The screw protruding from the side of the pump activates a bypass valve (or needle valve – see Figure 30) in order to regulate the airflow. The pump is supplied with a mounting bracket and four anti-vibration bolts (article 6 in Table 12 on page 61).

²³ Gas flowrate in standard temperature and pressure conditions (0°C, 1 bara).

²⁴ During this step, be sure to keep the pump head tight against the motor. Otherwise, the crankshaft and the bellows pump are no longer in the same plane, risking damage to them.

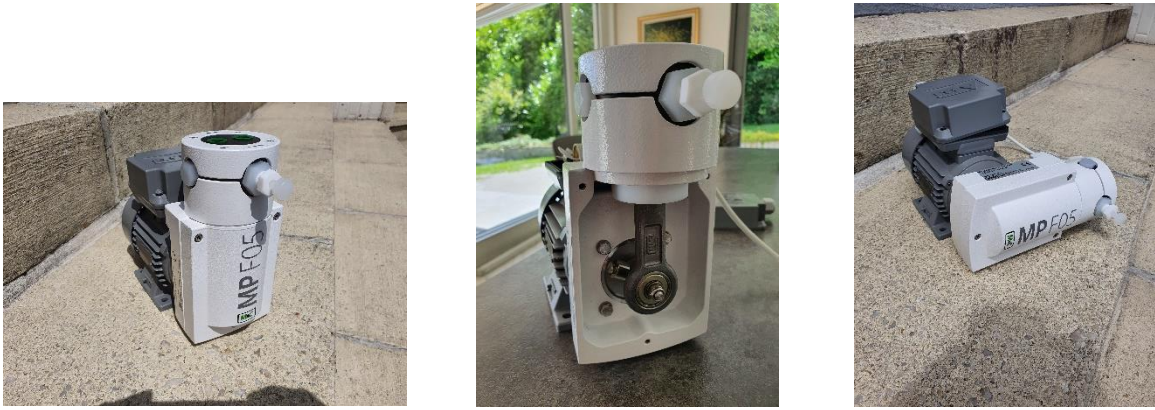


Figure 29 Air pump in its initial situation (on the left), unbolted, and rotated (on the right)

Source: own work

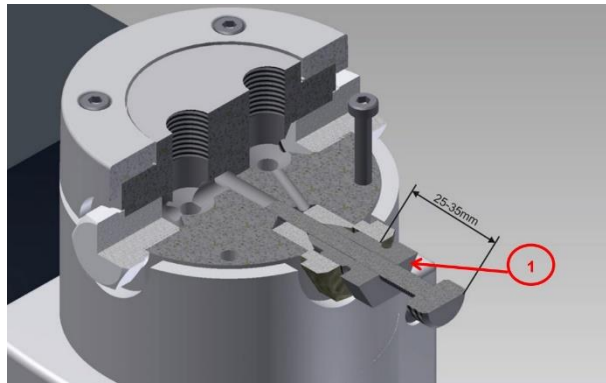


Figure 30 Operation of the needle valve of the M&C air pump

Source: M&C MP-F05/R instruction manual [20]

2.1.3 The particulate matter sensor (OPC-N3) box

The design of the OPC-N3 does not facilitate the connection to the tube system. However, it is provided with a seal around its air intake with four threads. This makes it possible to compress a 6mm diameter “through bulkhead” Swagelok connector against the OPC-N3 gasket. Unfortunately, all electrical junction boxes found on the market were a few millimetres too small in height to place the OPC-N3 in the same way as the CO₂ sensor (see 2.1.3 on page 64). This is the reason why a less robust box had to be used. Figure 35 shows how the tight connection to the OPC-N3 is performed, and Figure 31 shows how the cylindric box is fixed to the case panel.

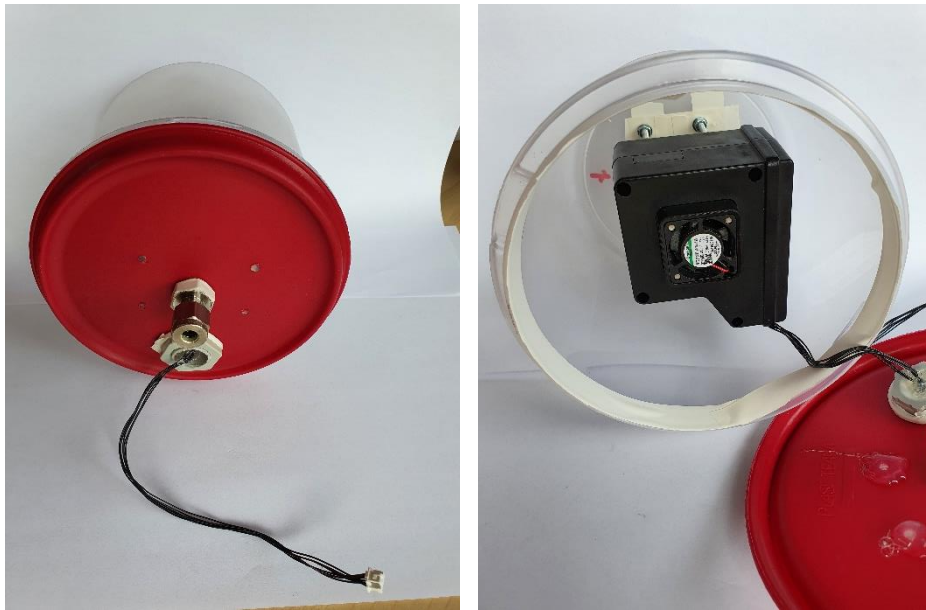


Figure 31 The OPC-N3 box

Source: own work

A Swagelok connector (article 9 in Table 12 on page 61) is compressed between the box and the OPC-N3 gasket via four threaded rods. The air leaving the sensor then ends up in the box and is then sucked through another Swagelok connector to the pump. Connections are sealed with Teflon, insulating tape and hot glue.

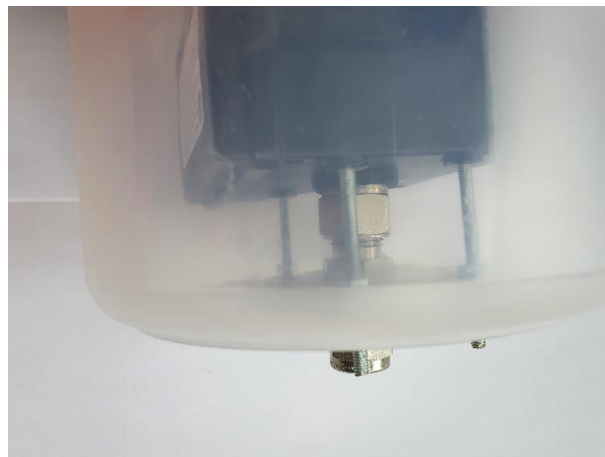


Figure 32 Connection of the tube system to the OPC-N3 via a Swagelok connector and four threaded rods

Source: own work

Finally, the sensor is fixed as close as possible and in line with the air intake to avoid any disturbance of the measurements caused by the potential accumulation of fine particles on the walls and the tubes' roughness. Finally, a mounting bracket was crafted from the remaining aluminium plates from the touchscreen cut-out. Figure 33 shows the OPC-N3 in its final position.



Figure 33 Fixing the OPC-N3 box to the bottom case panel

Source: own work

Due to its operation by means of a bellows and a crankshaft, the air pump generates a jerky airflow (see point 2.1.1 on page 62). The OPC-N3, using laser scattering to detect particulates, requires the most regular airflow possible. It is believed that the cover flexibility of the sensor box would absorb part of the air vibrations. In addition, placing the filter between the pump and the OPC-N3 helps to stabilize the airflow.

2.1.4 The CO₂ sensor box

The shape of the CO₂ sensor is absolutely not suitable for connecting a pipe system. For this reason, the sensor was placed in a small enclosure.

The CO₂ sensor has been placed in a small electrical junction box. The connection with the pipe system is made via an M&C PTFE connector (article 3 in Table 12 on page 61) which has been milled to fit into an electric gland cable (see Figure 35). In this way, the connector is clamped in the cable gland in the same way as an electric cable would have been. To increase the seal, Teflon has been added between the box and the cable gland. The electric cable passes through a plug initially designed to close the perforations of junction boxes. Instead, the plug was perforated, the cable passed and then sealed with hot glue.

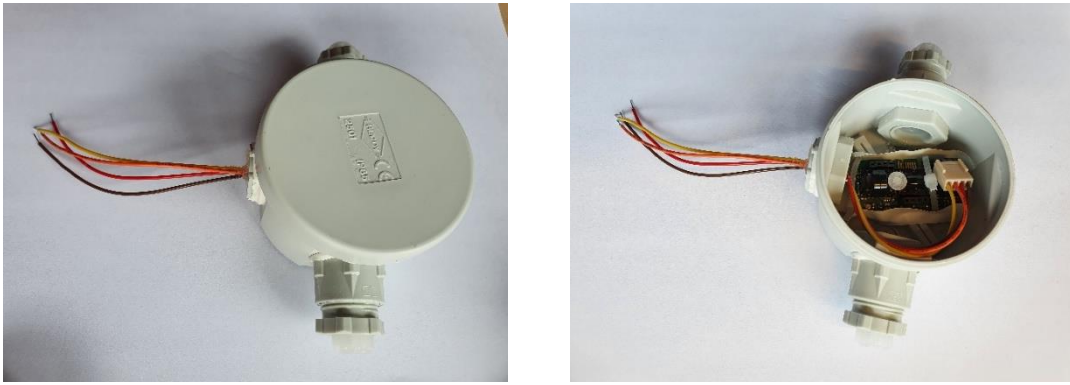


Figure 34 The CO₂ box

Source: own work

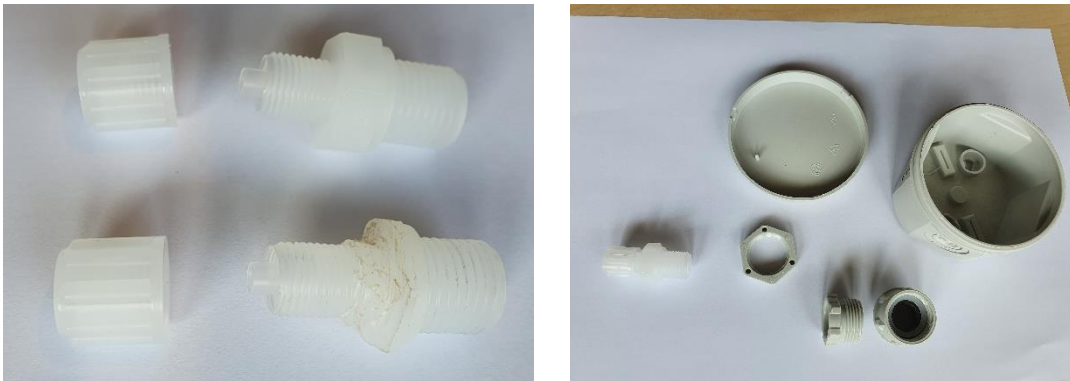


Figure 35 Shaping of the M&C connector and assembly of waterproof connectors

Source: own work

2.1.5 The Sensirion mass flow meter

The jerky movement of the air dramatically alters the flow rate readings by the flow sensor, which should therefore be placed in the same way as the OPC-N3 (see point 2.1.3). Since the flow sensor has been designed for medical purposes and pure air sampling, it has to be placed after the filter to prevent fouling the measuring cell and the internal filter. The problem of air fluctuation can be solved via the software by taking the average of several successive measurements.

2.1.6 The Alphasense 4AFE gas sensor

Gas sensors have been located at the end of the tube on the pressure side because they are not sensitive to airflow vibrations in opposition to the other sensors discussed above.

2.2 The electrical connection of all hardware components

The central computer and the air pump must be connected to a 220V AC source to operate. At the same time, several components such as the ADC and sensors must be connected

with each other and to power supplies of 5 VDC. All these connections require many wires, and this introduced errors (bad or unstable connections, moving wires generate noise) For that reason, a PCB board was designed that replaces most of the wires. At the same time, the power supplies must generate a stable voltage without any noise on the line. A maximum of connections was made in a junction box using Wago type connectors (article no. 5). A female connector (article no. 2) to be screwed into the Seacanairy male plug (article no. 3) provides the 220V supply. Articles can be found in Table 13.

Table 13 Electrical connections for power supply

Source: own work

| No. | Supplier | Reference | Name | Price per unit | Quantity | Price |
|-----|---------------|-------------------|--|----------------|------------------------------|------------|
| 1 | Cebeo Wavre | | 2 phase 1 earth + plug cable, 3 m length | 8.27 | 1 | 8.27 |
| 2 | RS Components | 934125100 CA 3 LD | Hirschmann Cable Mount Connector, 3 + PE Contacts 220V connector, cable side | 10.78 | 1 | 10.78 |
| 3 | | 932322100 CA 3 GS | Hirschmann Flange Mount Connector, 3 + PE Contacts 220V connector, Seacanairy side | 4.33 | 1 | 4.33 |
| 4 | Brico | | Junction box | | 1 | Negligible |
| 5 | | | Wago 5 way connectors Connect 220V wires | | 4 | 6.79 |
| 6 | Cebeo | | 220V one strand electric wire Pull cables between components | | 5m ground, 5m blue, 5m bruin | 8.27 |
| 7 | | | Traco power TPC-030-105 Supply 5V (max 4A) to the Raspberry Pi and the sensors | | 1 | Unknown |

| No. | Supplier | Reference | Name | Price per unit | Quantity | Price |
|-----|----------|-----------|--|----------------|---------------------------------------|-------|
| 8 | | | <p>Tokin noise filter LF205A 250V 5A</p> <p><i>Electrically isolate the M&C air pump to the other components</i></p> | | 1 | |
| 9 | | | <p>Lugs</p> <p><i>Earth the aluminium plates, and make the connection of the pump to the TOKIN noise filter</i></p> | | 2 rounded 5 female disconnects [5] | |

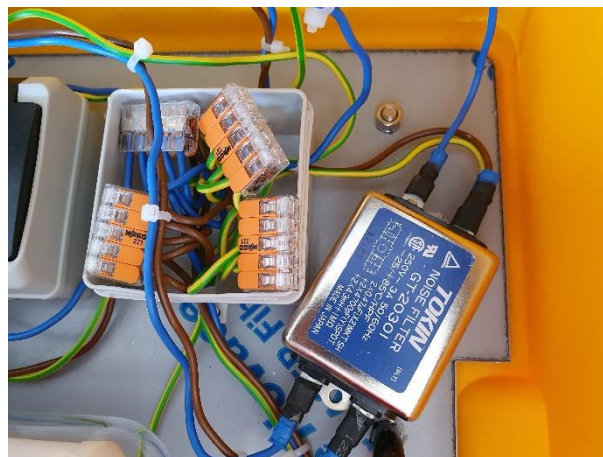


Figure 36 Picture of the 220V derivation box (on the left), and the Tokin noise filter (on the right)

Source: own work

2.2.1 Electric noise on the 220V line

As explained in point 2.1.1 on page 62, the pump generates some electrical noise when running, which interferes with the proper functioning of the OPC-N3. The problem was solved by connecting a noise filter (article 8) between the pump and the 220V power supply. The relay operated by the central computer has been positioned upstream of the filter so as to prevent it from being permanently under voltage. The connections are made by means of female lugs (article 9), protected by heat-shrink tubing. A schematic of the connections is to be found in Figure 37 as well as in Annexe 3 on page 117. Do not confuse the line side (voltage source) with the load side (pump).

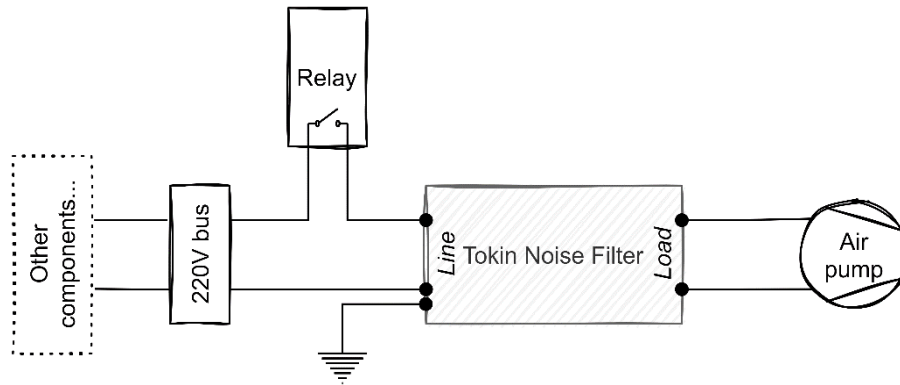


Figure 37 Schematic of the wiring of the Tokin noise filter on the M&C air pump

Source: own work, using draw.io

3 Central computer

The central computer is composed of three main parts: the Raspberry Pi (the computing unit), the analogue to digital converter, and the custom printed circuit board, making possible the connection of those two parts to the various sensors. Those three components are compatible with HAT (hardware on top) and form a single unit, as shown in Figure 38.

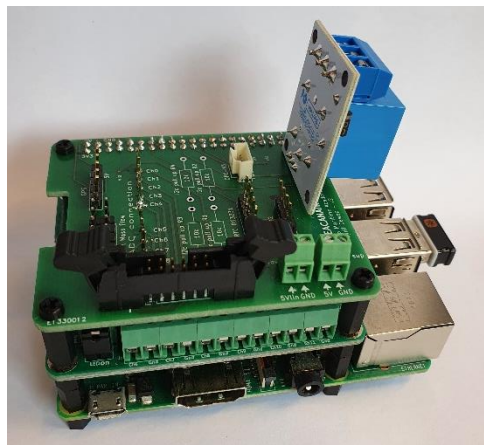


Figure 38 Central computer unit (from bottom to top: Raspberry Pi, Pi16-ADC, custom printed circuit board)

Source: own work

3.1 The Raspberry Pi

A central computer manages the Seacairy measuring system. The device used is a Raspberry Pi 3B+. It is a single piece nano-computer from the ARM family designed for do-it-yourself electronics. The computer is equipped with an Ethernet connection, Wi-Fi, Bluetooth, four USB, audio 3.5 mm jack, CSI camera interface, DSI connector for the official screen, a MicroSD socket for storage, and HDMI. These are standard functionalities that can be found on any recent computer. In addition, the Raspberry Pi has two connector rows, each with 20

pins that are directly linked to the BCM2837 chipset. Those General Purpose Input/Output (GPIO) can be used as input or output to communicate with any device or sensor, either controlled manually by any self-made software or automatically using designated pinout for particular purposes [19].

3.2 The Analog to Digital Converter

The Raspberry Pi can communicate with digital sensors but cannot process any analogue signals. Therefore, an Analog to Digital Converter is necessary in between the gas sensor and the Raspberry Pi. The device used is the PI-16ADC from Alchemy Power. It provides a 16-bits conversion, which leads to a reading accuracy of 38.1 microvolts [1]. In addition, it is compatible with a HAT add-on board, as the ADC, that can easily be plugged on top of the Raspberry Pi, forming a single unit with the Raspberry Pi.

Table 14 Inventory of the Central Computer

Source: own work, Master thesis of Lukas Van der Borghet [37]

| No. | Piece No. | Name | Quantity | Price |
|-----|------------------------|-------------------------------|--------------|-------|
| 1 | Raspberry Pi 3B+ | Raspberry Pi 3B+ | 1 | 50.00 |
| 2 | Alchemy Power PI-16ADC | Analogue to Digital Converter | 1 | 50.00 |
| | | Custom Printed Circuit Board | By two units | 53.36 |

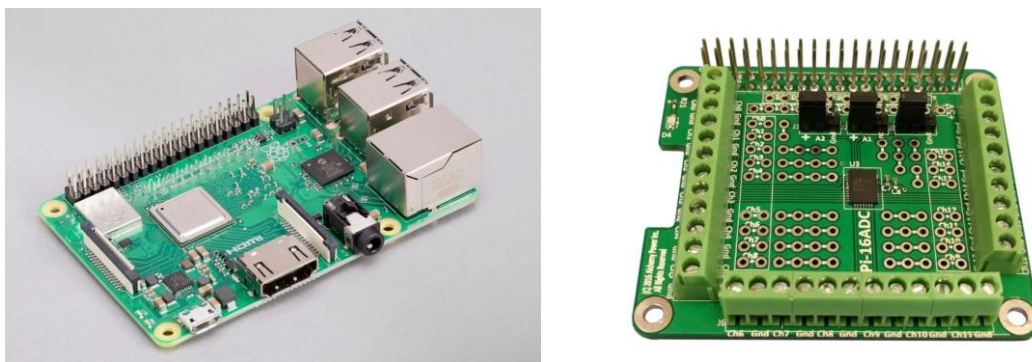


Figure 39 Picture of the Raspberry Pi 3B+ (on the left) and the PI-16ADC (on the right)

Source: Raspberry Pi website [19] and Alchemy Power website [1]

3.3 Printed Circuit Board (PCB)

During the Seacairy development phase, a breadboard was used to test all the electrical connections between the sensors and the central computer. Figure 40 illustrates the complexity of the connections and the tangle of jumper cables (fast prototyping) after connecting the sensors accordingly. This kind of fragile connection takes up space, is subject to electrical noise and easily

disconnect. The production of a printed circuit allows all the required connections to be condensed into a tiny plastic plate. The following points summarize the steps followed to build the Seacanairy printed circuit board using the software KiCad.

Table 15 Inventory of the printed circuit board

Source: own work

| No. | Supplier | Reference | Name | Price per unit | Quantity | Price |
|-----|---------------|-----------------|---|----------------|---------------------|-------|
| 1 | Eurocircuits | | Custom Printed Circuit Board | | 2 | 53.36 |
| 2 | RS Components | SSW-101-02-T-S | Samtec, SSW 2.54mm Pitch 1 Way 1 Row Straight PCB Socket, Through Hole <i>Single female header for soldering on the PCB into ADC connections</i> | 0.43 | 2 packs of 10 units | 8.60 |
| 3 | | 1725656 | 2 way PCB vertical mount terminal, 2.54mm <i>Terminal block for 5V and 5V-linear power supply</i> | 1.81 | 1 pack of 5 units | 7.48 |
| 4 | | IDSD-04-D-10.00 | Samtec Slim Body Double-Row IDC Socket Assemblies, 0.100" Pitch <i>2 row 8 way cable for the flow meter sensor</i> | 3.98 | 1 | 3.98 |
| 5 | | SSQ-120-03-G-D | Samtec, SSQ 2.54mm Pitch 40 Way 2 Row Straight PCB Socket, Through Hole <i>40 way 2 row female header with long legs for future GPIO use</i> | 8.70 | 1 | 8.70 |
| 6 | | 15133-0606 | Molex Pico-Clasp OTS Wire to Board Cable Assembly 1 Row, 6 Way, 600 mm length <i>Cable between the OPC-N3 and the printed circuit board</i> | 5.45 | 2 | 10.90 |
| 7 | Mantec Namur | VMA406 | 5V relay module <i>To operate the 220V air pump</i> | 6.90 | 1 | 6.90 |
| 8 | | TSW-140-09-G-S | Pin Header 40 way 1 row, 18 mm, golden <i>Multiple purpose: CO₂ sensor connection on the sensor side and PCB side, male header for ADC connection, male header for the Real Time Clock, male header for the GPS</i> | 1.02 | 1 | 1.02 |

| No. | Supplier | Reference | Name | Price per unit | Quantity | Price |
|-----|----------|-------------|--|----------------|--------------------|-------|
| 9 | | 501331-0607 | MOLEX 1.00mm Pitch, Pico-Clasp PCB Header, Single Row, Vertical, Surface Mount, 6 Circuits <i>Connection of the OPC-N3 on the printed circuit board</i> | 6.61 | 1 pack of 10 units | 6.61 |
| 10 | | 15133-0603 | Pico-Clasp 1 row 6 way 30 cm cable | 6.61 | 4 units | 26.45 |
| 11 | | | Connector with cable for PCB, 2.54 pitch, 4 way, 1 row <i>For connecting the CO₂ sensor through its 2.54 pitch female header</i> | 0.95 | 2 | 1.90 |
| 12 | Amazon | | Yuhtech Hex Spacer Screw Nut Assortment (M2.5) <i>To fix the different parts of the mainframe together</i> | 9.06 | 1 | 9.06 |
| 13 | | | POPESQ® wire-to-board connector 4 way 1 row, 20 cm length | 5.39 | 2 packs of 3 units | 10.78 |

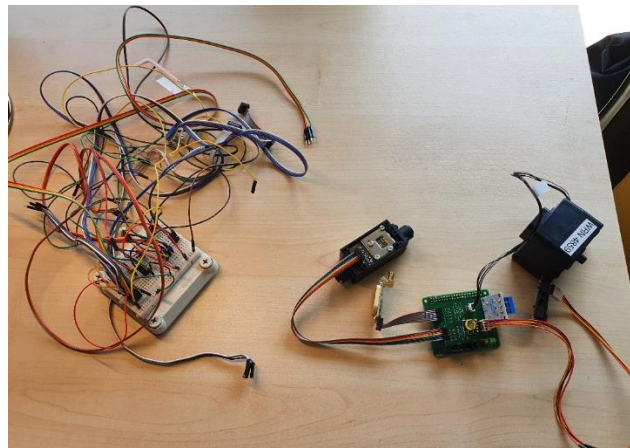


Figure 40 Overview of the wiring of the first prototype (on the left side) and overview of the wiring of a similar system using the PCB-board (on the right side)

Source: own work

3.3.1 General procedure for designing a PCB

The tracing of a printed circuit board is achieved in several stages: firstly, the drawing of the electrical diagram (also called schematic), the attribution of a footprint to each symbol, the acquisition of the characteristics of the printer where the circuit board will be printed, and finally the tracing of the printed circuit.

The first step consists of schematically drawing all the connections of our breadboard into one single schematic. This diagram is composed of symbols representing one specific component: a resistor, a connector, or a terminal block. For connectors, symbols are generics and disregard the physical properties of the connection (distance between pins, through-holes, or surface mounted connectors). After inserting all the required symbols, wires are pulled to make the desired connections. Instead of drawing lines in all directions, names are given to each wire, knowing that all wires having the same name are interconnected.

Once the schematic is drawn, footprints are linked to each symbol. The footprint is the name given to the physical shape and geometry of a symbol present in our schematic once mounted on the printed circuit board (see example in Figure 41). Through this step, each symbol gets physical properties, such as the fixing type (surface mount or through holes), the spacing between the pins, the number of rows, and the numbering of the pins (pair, odd, clockwise, or counterclockwise).

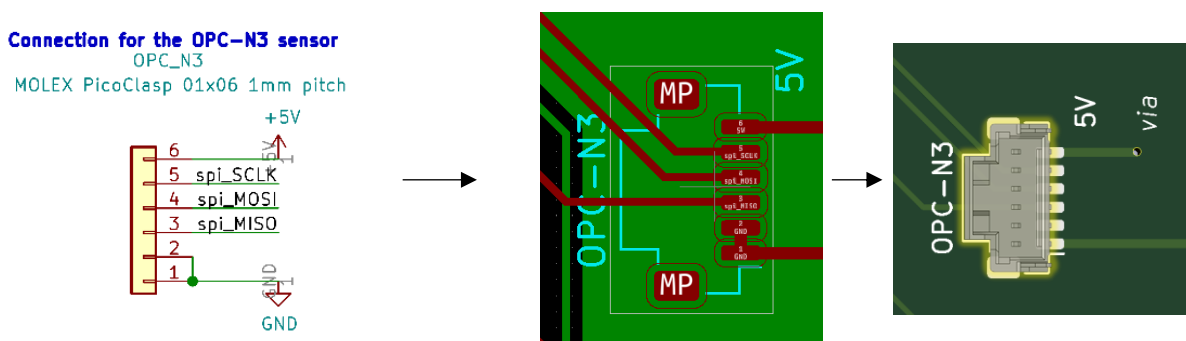


Figure 41 Comparison of the symbol on the schematic with the footprint on the printed circuit board

Source: own work, using KiCad

The circuit board drawing starts by studying the printer restrictions such as the minimum track width, hole diameter, via holes size, and the number of available layers. Then, those parameters are inserted in KiCad, which will care to respect them.

Sadly, KiCad does not draw the PCB automatically. All the footprints need to be manually placed where wished and required connections to be manually drawn. Since it is a print (two dimensions), no line can cross, as this would generate a short circuit. The position of the footprints must avoid as much as possible the crossing of lines. As a last resort, vias are used: perforations are made, allowing the electricity to pass from one layer to another. As a result, it takes a few hours to find the best drawing (see Figure 42).

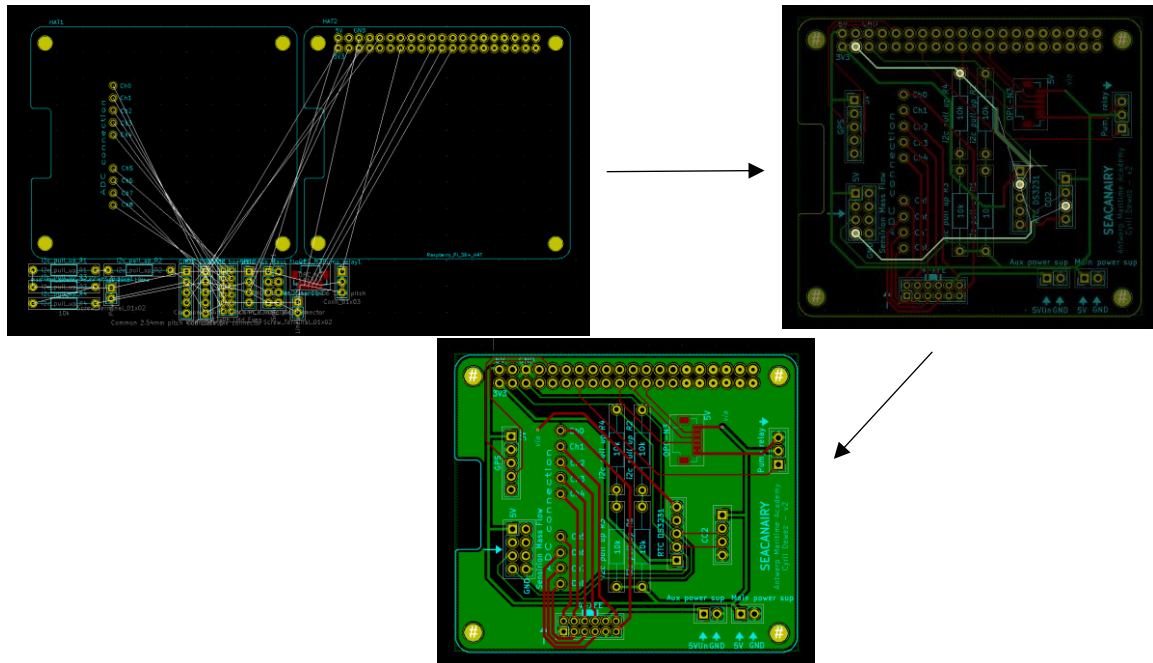


Figure 42 Positioning of the footprints and tracing of the electric lines

Source: own work, using KiCad

Finally, the required components are soldered to the PCB. Table 15 on page 72 lists all the components necessary for creating the PCB. See Annexe 3 and Annexe 4 on pages 117 and 119 for design details.

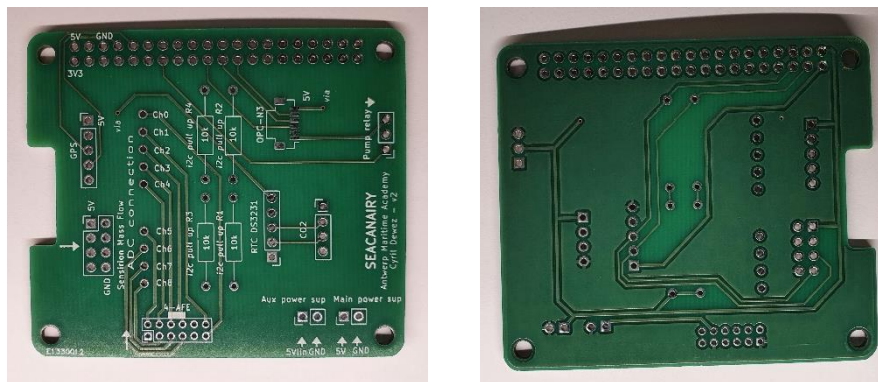


Figure 43 Printed circuit as supplied by Eurocircuits

Source: own work

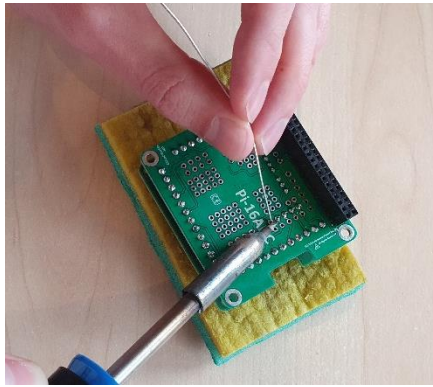


Figure 44 Welding the connectors on the custom PCB

Source: own work

3.3.2 Seacanairy wiring

The global electrical wiring of the Seacanairy is shown in Annexe 3 on page 117. Sensor specific wirings are explained accordingly in Chapter 1 from page 3.

3.3.3 The connection between the Analog to Digital Converter (ADC) and the custom circuit board

The main difficulty in designing the printed circuit is the connection between the printed circuit board and the ADC located one floor below. In order to remove any electrical cable, it was necessary to match the perforations in the printed circuit board with the perforations in the ADC. First, the three-dimensional file provided by the ADC manufacturer (Alchemy Power) has been converted to a KiCad footprint using FreeCAD with the KiCadStepUp add-in. Next, several shapes' projections and clippings have been carried out to constitute the different layers required by KiCad. Then, the created footprint has been transferred to the KiCad library. Finally, the perforations have been added, and the pads numbered in the same logic as the symbol (the difference between symbol and footprint is explained in more detail in point 0 and Figure 42 on page 75).

The ADC has been designed with a pitch (distance between the perforations) of 3 mm instead of 2.54 (standard). This kind of part being untraceable, single headers (item number 2 in Table 15 on page 72) were purchased and soldered where required.

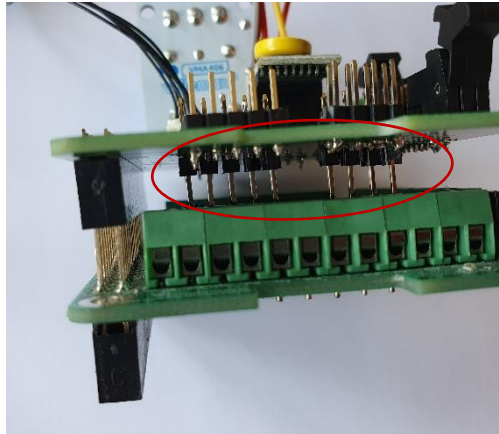


Figure 45 Connection between the printed circuit board and the ADC

Source: own work

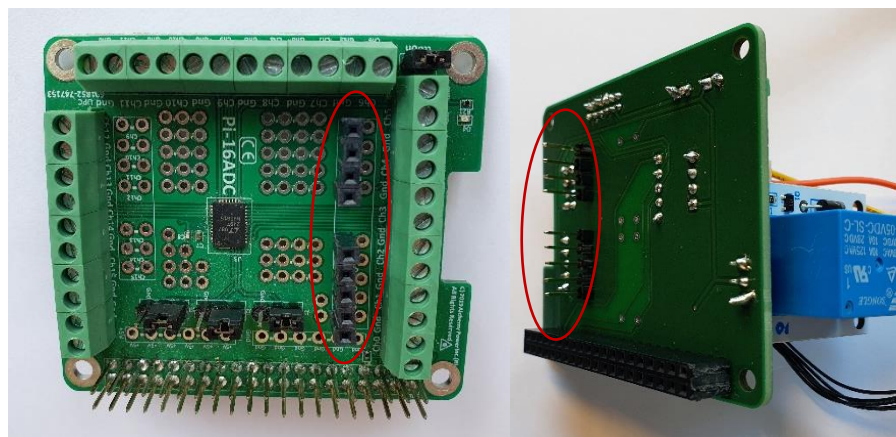


Figure 46 Female header on the ADC and male header on the printed circuit board

Source: own work

3.3.4 Tips for a successful printed circuit

- After placing the footprints and before starting drawing in KiCad, check that it is physically possible to place all the necessary connectors on the board (avoid putting two connectors too close).
- Make sufficiently large perforations. This facilitates the desoldering of components.
- Place sufficiently large pads²⁵. This makes it easier to weld through holes components because the weld has a larger surface to deposit.

²⁵ Name given to the conductive surface around a perforation on a PCB on which the solder is placed.

- Use through holes perforations as vias (to create a connection from one layer to another). This allows for decreasing the number of perforations on the board and spare place on the PCB.
- Place the tracks on the opposite layer from which the welds will be applied. This helps prevent damage to the tracks when soldering the connectors.
- Before soldering, tin the tip of the soldering iron by melting some solder on it. This makes it possible to have a liquid contact between the soldering iron and the printed circuit, and therefore better heat conduction. Then, when the circuit pad is hot, the solder strand can be approached.
- Keep the soldering iron temperature as low as possible to avoid damaging the circuit board (around 250 °C).

Chapter 3

Setting up the development environment on a stand-alone computer

The previous chapter showed how the sensors were connected by tubes so that the same air passes over all the sensors. It also showed how all the electronic components were connected to a voltage source. To allow an actual communication between the central computer (i.e., Raspberry Pi) and all the connected sensors, a software need to run on the central computer. The software was written on a stand-alone computer, and then regularly transferred to the Raspberry Pi for execution and tests. Software and libraries are executed in a virtual environment so that their operations are detached from the system. The software aims to connect all the sensors in a single frame, get the measurements simultaneously, store all the data in a single database, and provide real-time information to the operator on a screen.

The central computer used is designed to work with Linux and the Python language. Python is a language known for its simplicity, rich in features, reliability and efficiency. Before starting to write our Python code, a series of steps are necessary to set up a software development environment on a stand-alone computer in which we will be working during the coding process. The configuration of a comfortable environment is essential for realising such a complex project with no initial knowledge of Python programming. The first part of this chapter explains how to set up the integrated development environment on its personal computer. The editing software, the automatic backup system, and the addition of libraries to the editing software are explained. The second part of this chapter deals with the Seacanairy central computer. It is concerned with the remote connection to the central computer, the file transfer, how to update the central

computer, the Python virtual environment, as well as the procedure for testing our software as it is being written.

1 Set up the development environment on a personal computer

1.1 Development software – PyCharm

The first step is to find the software to code with. PyCharm has been selected among the wide variety of IDE (integrated development environment), because it is one of the most accessible IDE to start coding with Python. Similar to Words capability to highlight mistakes during typing, PyCharm incorporates Python and other components for continuous code monitoring. Hence, PyCharm can point out errors, suggest simplifications, increase readability, or invite to add comments. Along the way of writing the code, PyCharm suggests new tips and tricks to the user to improve his code. Thus, the user can improve his skills in Python while working on his project. Figure 47 shows a screenshot of the main PyCharm display. The yellow lamp on the left indicates a little trick that the software suggests to the user. As an example, the improvement shown is the addition of a line break at line 267 to avoid text exceeding the screen size. PyCharm IDE has a free and premium version. The free one is sufficient for the Seacanairy purpose.

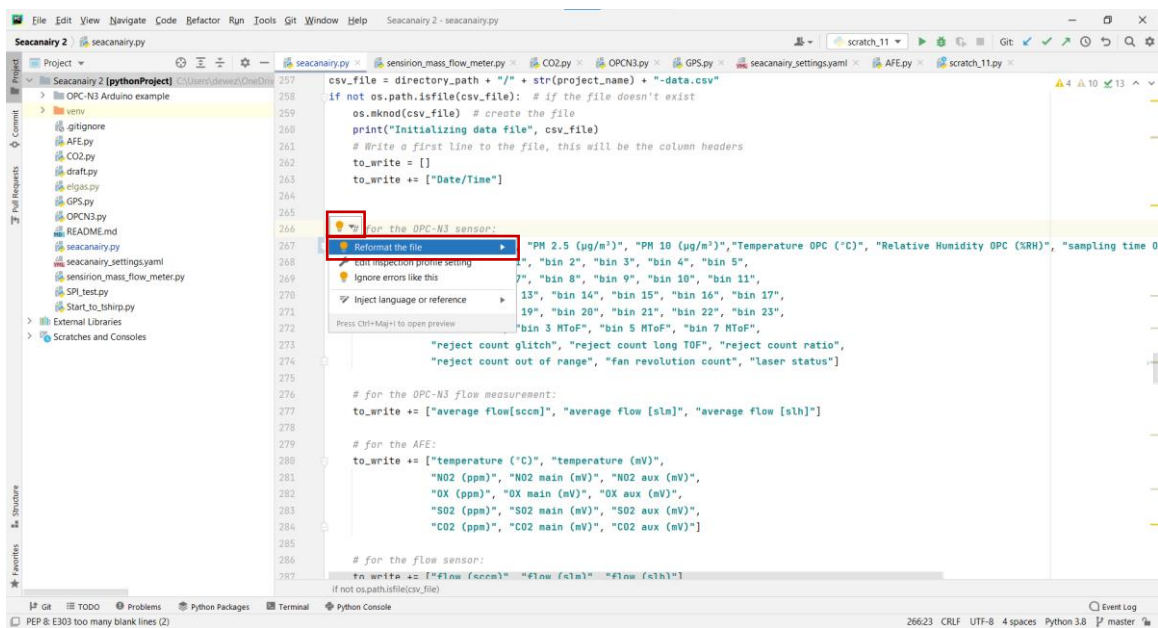


Figure 47 PyCharm screenshot

Source: own work

1.2 New project creation

Once PyCharm has been installed, create a new project. Indicate the location where your code (what you are typing) will be stored. In the Virtualenv section, indicate the location where the virtual environment will be stored. Do not store the virtual environment in the same folder as the code. This creates a lot of files and confusion between code and the system files. The use of a virtual environment allows the user to work on different projects using different Python versions, different library versions and Python interpreters. Figure 48 shows a PyCharm IDE screenshot. After the project is created, the IDE starts a Python script example named `main.py`.

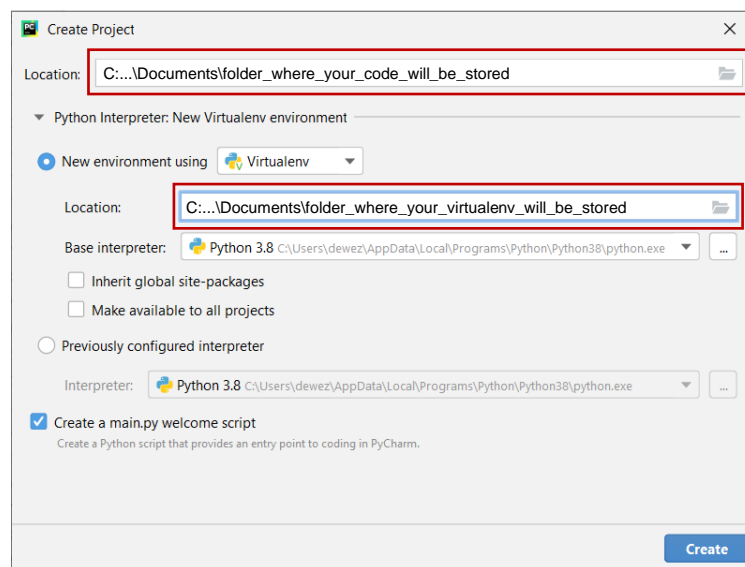


Figure 48 Create a new project in PyCharm

Source: own work

1.3 Git repository and GitHub account

It is common sense to keep a backup of the software. Best practice dictates that during the whole Seacanairy development process, it is necessary to create a repository on GitHub and to push any new version of our code on the cloud. GitHub is a free cloud platform for software backup, file sharing, and collaboration. PyCharm is fully compatible with GitHub after the installation of Git.

Start by creating an account on github.com. Then, install the Git package via git-scm.com/downloads. Once Git is installed, restart PyCharm and open the PyCharm settings (File/Settings). In Version Control, in Git, check that PyCharm has well detected the path to the Git executable. If the path has not been detected, fill it in yourself. The path should look like this: `C:\Program Files\Git\cmd\git.exe`. Figure 49 shows the procedure and menu to

Setting up the development environment on a stand-alone computer open to reach the settings. Once the plugin has been successfully linked, connect Git to your GitHub account. On the same window, open GitHub, press + on the top and Log In via GitHub. Refer to Figure 50 for more details.

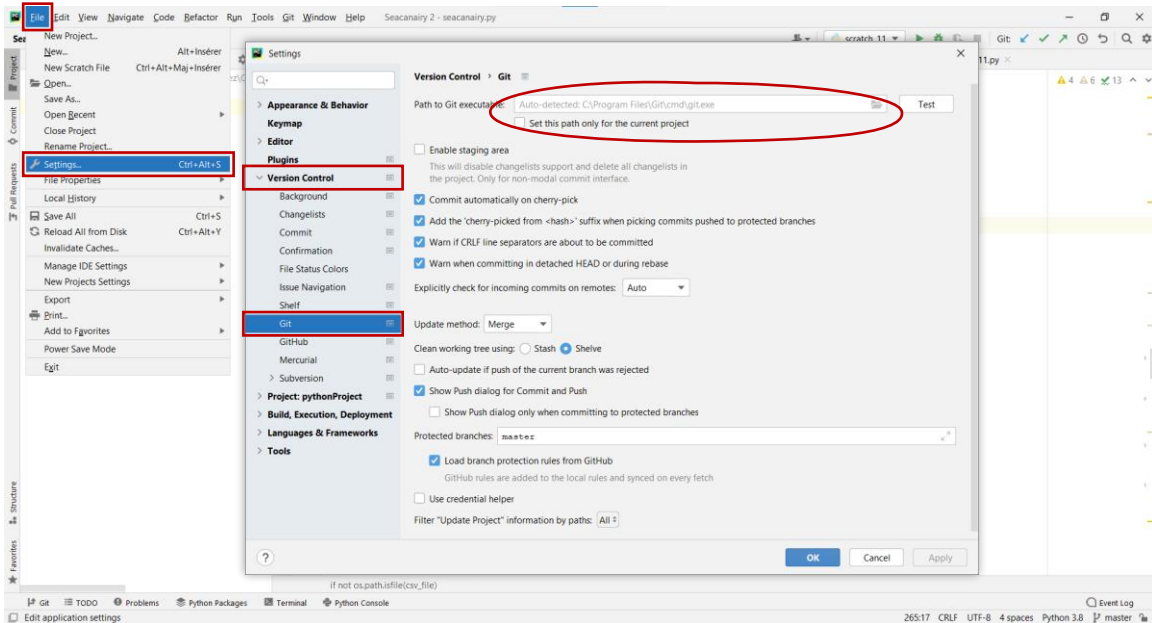


Figure 49 Git incorporation to PyCharm

Source: own work

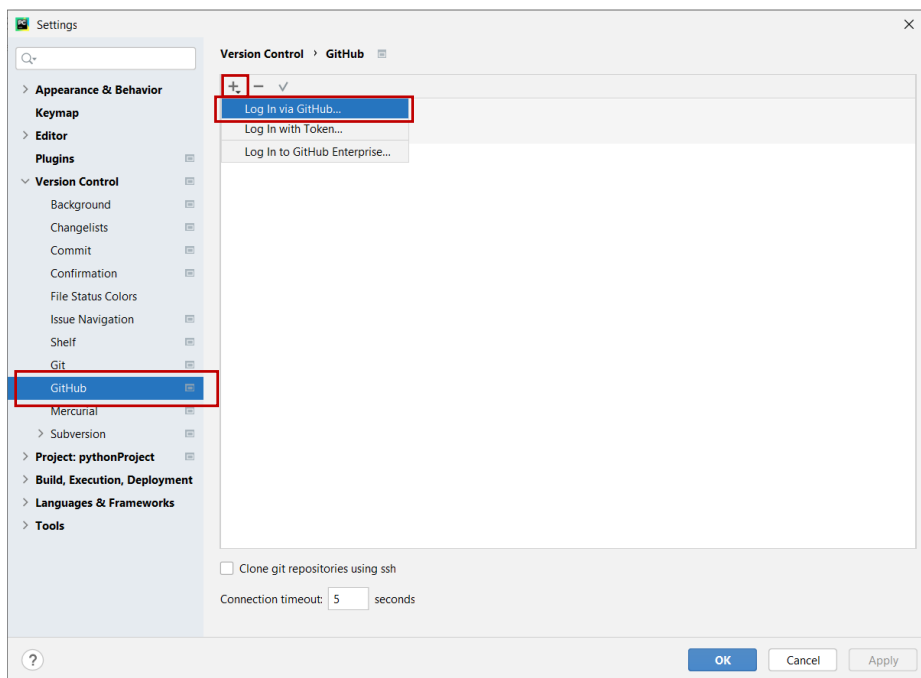


Figure 50 Log in GitHub using PyCharm

Source: own work

Once PyCharm is configured to communicate with the GitHub servers, we can create a repository. This is the name given to the folder which will be synchronized with the cloud. In PyCharm, open the vcs tab and create a Git repository. Now, share this repository on

your GitHub page. In VCS, click on share project on GitHub. Give your new repository a name and a short description. Once this step is done, you see a new project on your GitHub page as in Figure 53.

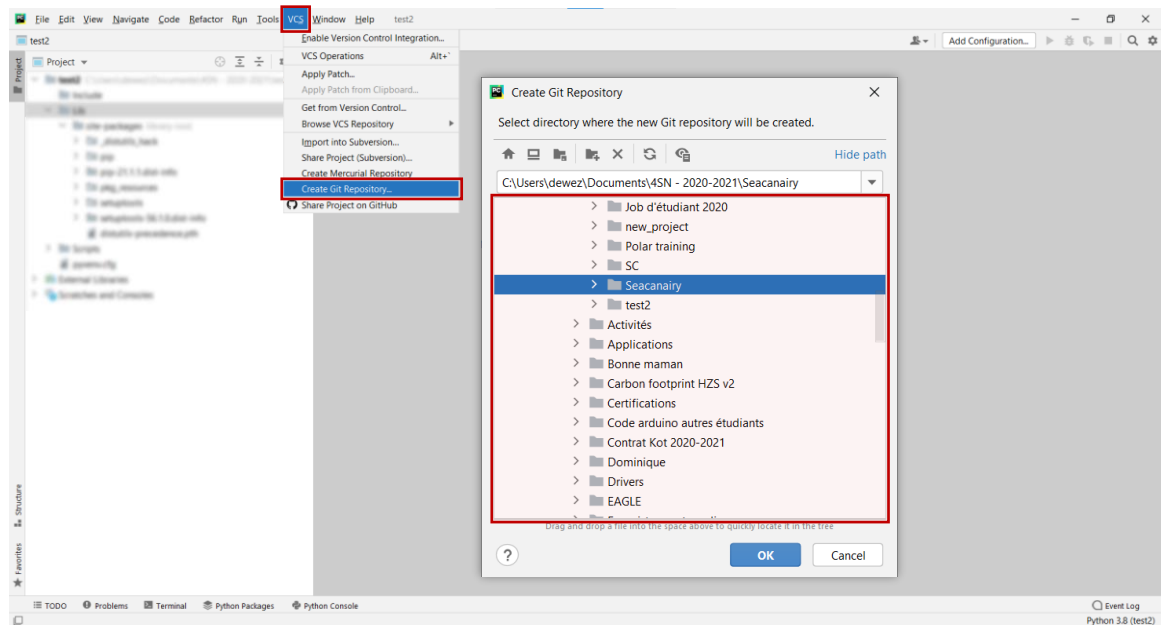


Figure 51 Create Git repository through PyCharm

Source: own work

1.4 Commit and Push files to GitHub

Every time the user makes a series of changes, they should be posted immediately to the server by pressing commit. Select the file you want to update and write a comment. The comment should shortly describe what changed in the new version. Once you commit to all the desired files, push them to the cloud. Refer to Figure 52 for more details concerning the procedure. Stored and updated files are also visible on the GitHub web page, as shown in Figure 53.

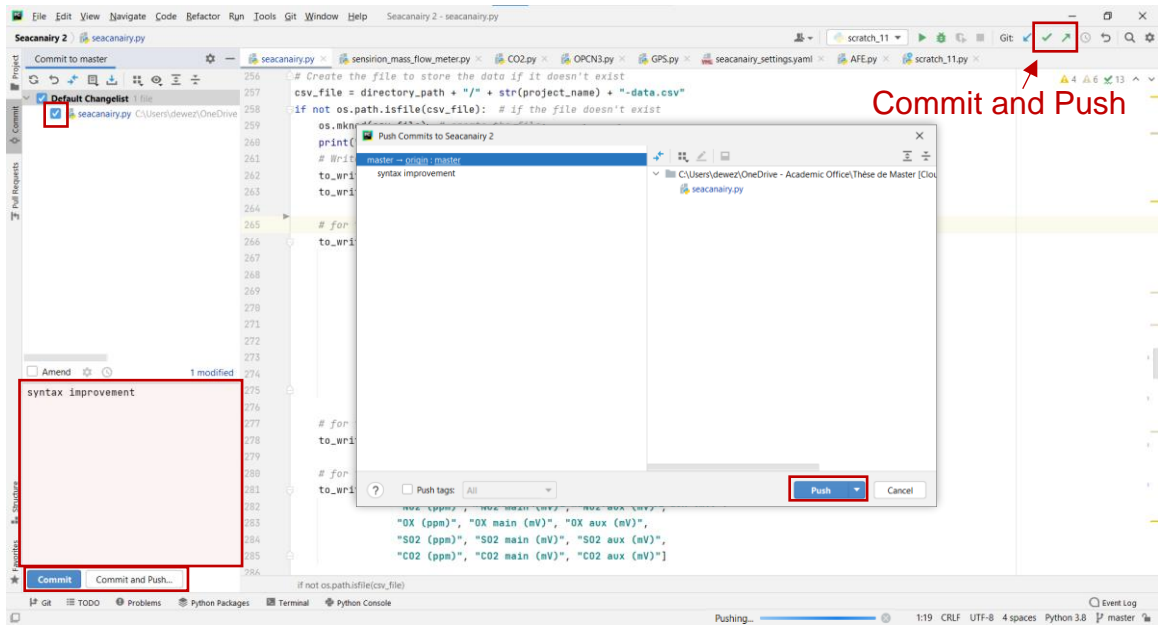


Figure 52 Commit and Push changes to GitHub

Source: own work

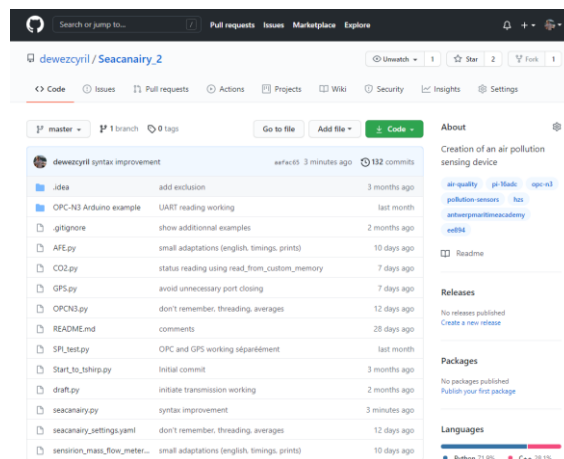


Figure 53 GitHub repository example

Source: own work

1.5 Libraries installation

While coding in Python, for the Raspberry Pi, libraries are used. These libraries are sets of predefined useful functions. The required libraries have to be installed in PyCharm. In this way, the IDE will be able to check the code and help us. To install a new library in PyCharm, open File, then open Settings, click on the Project tab and click on Python Interpreter. Then, use the button + and – on top of the page to add and remove libraries. The procedure is shown in Figure 54. Hereafter is a list of the required libraries:

- MySQL-connector

- Progress
- Pyserial
- PyYAML
- Raspberry Pi.GPIO
- Smbus2
- spidev

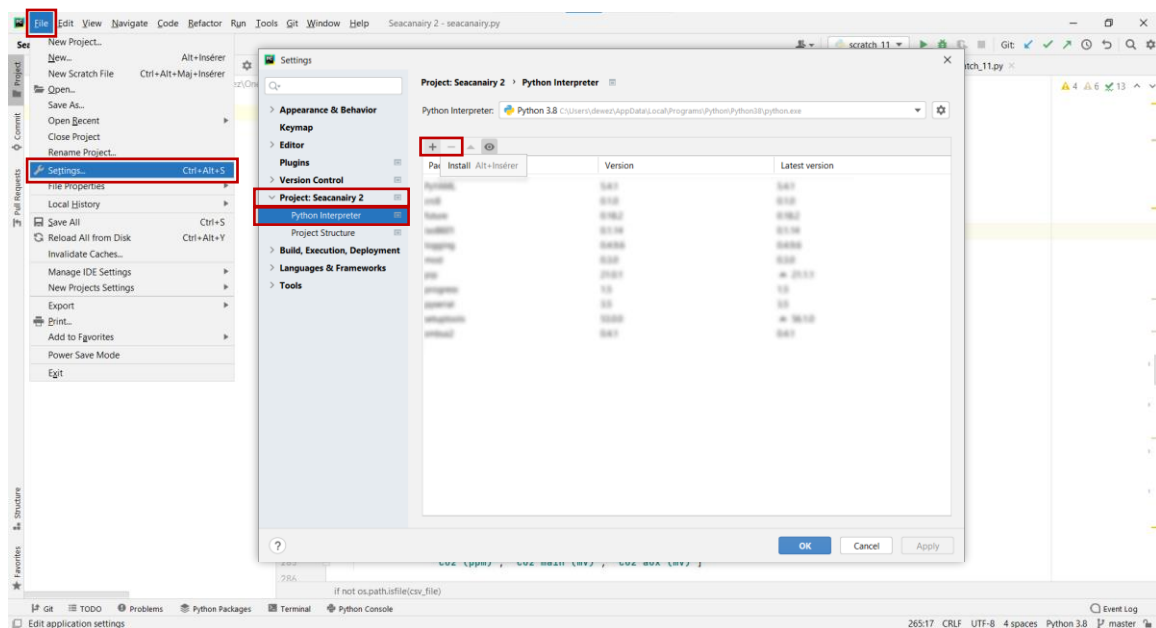


Figure 54 Install libraries on PyCharm

Source: own work

1.6 Connect to the Raspberry Pi using TeamViewer

TeamViewer is an easy software solution allowing remote control of any computer from any device. It is used for debugging in informatics, the internet of things, and remote access. In contrast to SSH (Secure Shell), TeamViewer works from any connection (either local or worldwide), which requires a local connection (computer connected to the same router as the Raspberry). Download TeamViewer on your computer from the official website ([teamviewer.com](https://www.teamviewer.com)). On the Raspberry Pi, open TeamViewer using the shortcut on the Seacanairy central unit desktop, using either the touchscreen or a USB mouse plugged in the Raspberry Pi USB). Meanwhile, on your PC, insert the partner ID and type the password when required. Current Seacanairy central computer partner ID and password are written in Table 16.

Table 16 Raspberry Pi TeamViewer ID and Password

Source: own work

| Partner ID | Password |
|------------|------------|
| 1612778925 | se@c@n@iry |

1.7 Transfer files from or to the Raspberry Pi

To transfer files from your computer to the Seacanairy central unit and in the opposite direction, connect to the Raspberry Pi as explained in section 1.6. On the top of the computer screen, click the TeamViewer banner and open **File Transfer**. A window will open, showing side by side the PC storage and the Raspberry Pi storage. Next, select a file on the right or left and click on the corresponding transfer button, either **send** or **receive**. Multiple files can be transferred at once, holding down **ctrl** or **shift**.

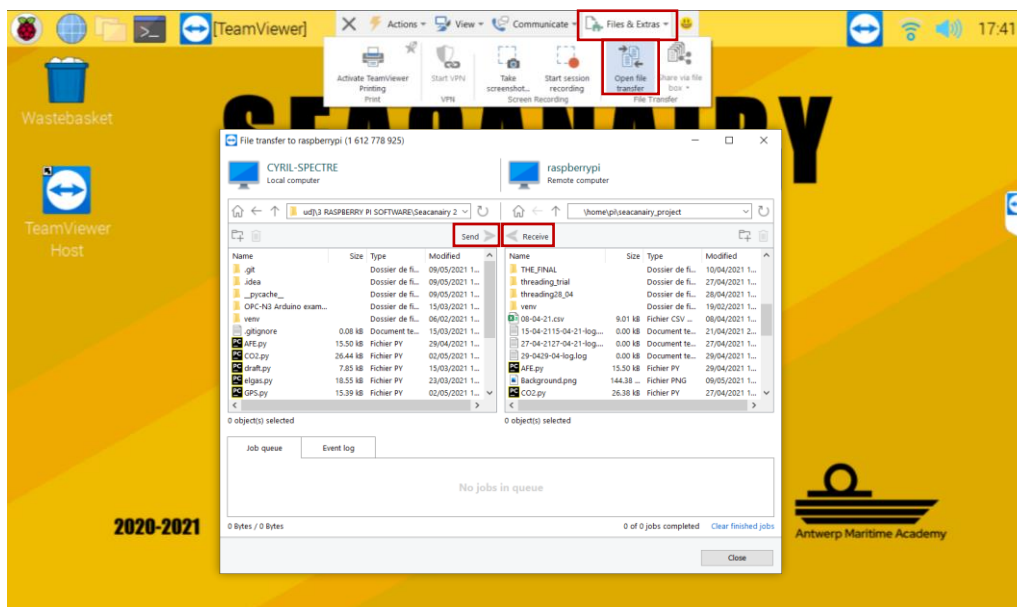


Figure 55 File transfer from/to the Raspberry Pi

Source: own work, screenshot of the central computer screen, taken through TeamViewer

2 Set up the development environment on the Seacanairy central computer

2.1 Update the Raspberry Pi

As modern electronic devices connected to the internet, they are suffering from vulnerabilities and bugs. Raspberry Pi OS, software's and other libraries are in constant evolution. The procedure to update the Raspberry Pi is described [36]. A summary:

- Update the system's package list: `sudo apt update`
- Upgrade the package list to the latest version: `apt list --upgradable`
- Run all the upgrades in the list. During this last step, do not disconnect the Raspberry Pi power supply. Do not worry about screen behaviour which can become black for a while. The display will return when the update is finished.
`sudo apt full-upgrade`, confirm via `Y`

2.2 The virtual environment on the Raspberry Pi

The virtual environment stores all the libraries used in one project in one separate folder. That way, the user can work on different projects with the same device using different libraries and versions, for example.

2.2.1 Create a Virtual Environment

This point aims to explain how to create a new virtual environment on the Raspberry Pi [16].

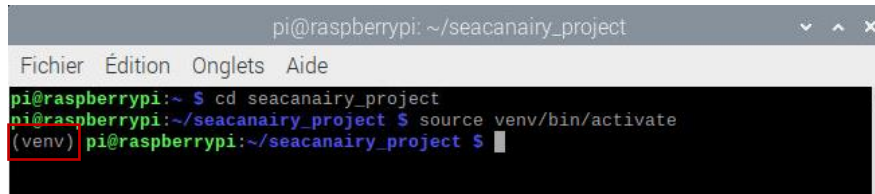
- Create a folder in the storage where Python codes, virtual environment, and data files will be stored altogether.
- Open a new terminal on the Raspberry Pi and open the folder created before using `cd` followed by the folder name. Execute `cd seacanairy_project`
- Execute the following function to create the virtual environment. Let the process work until it is finished. Execute `python3 -m venv ./venv`

2.2.2 Virtual environment activation

Before any Python-related action in the command shell, activate the virtual environment. The following steps are illustrated in Figure 56 [16]. Refer to point 3 on page 92 for console tricks.

- Open the folder in which your virtual environment is saved using the `cd` command. Execute `cd seacanairy_project`
- Activate the virtual environment. Execute `source venv/bin/activate`

- The virtual environment is now active. Subsequent commands entered as `pip` or `python` will be executed in the virtual environment. To leave the virtual environment, type `deactivate` and press enter. Note the presence of `(venv)` on the left of the green text.



```
pi@raspberrypi: ~/seacanairy_project
Fichier  Édition  Onglets  Aide
pi@raspberrypi:~ $ cd seacanairy_project
pi@raspberrypi:~/seacanairy_project $ source venv/bin/activate
(venv) pi@raspberrypi:~/seacanairy_project $
```

Figure 56 Activate the virtual environment on the Raspberry Pi

Source: Own work, a screenshot of the central computer screen, taken through TeamViewer

2.2.3 Activate the virtual environment in Thonny Python IDE

Thonny Python IDE is the built-in IDE for Python on the Raspberry Pi. It is not as powerful as PyCharm (see page 80), but it is an easy way to make small modifications to a code during development. By default, Thonny Python IDE uses the built-in Python 3 version. To run our code based on our virtual environment, some changes must be made in the settings. To do this, open Thonny Python IDE using a USB mouse or use the touchscreen as shown in Figure 57. Then, follow the procedure, also explained schematically in Figure 58.

- In `tools`, open `options`.
- Open the `interpreter` tab. Select `Alternative Python 3 interpreter` or `virtual environment`.
- Below, indicates the location of the `python` file in the virtual environment created at point 2.2.1. The path should look like `.../venv/bin/python`. Click on the `'...'` on the right to navigate through the folders.
- Click on `OK` and restart Thonny Python IDE. After the restart, you should see the new interpreter path on the right bottom of the window.

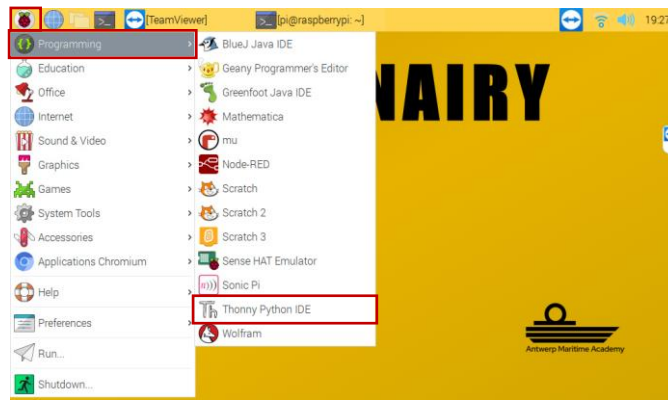


Figure 57 Opening Thonny Python IDE

Source: own work, a screenshot of the central computer screen, taken through TeamViewer

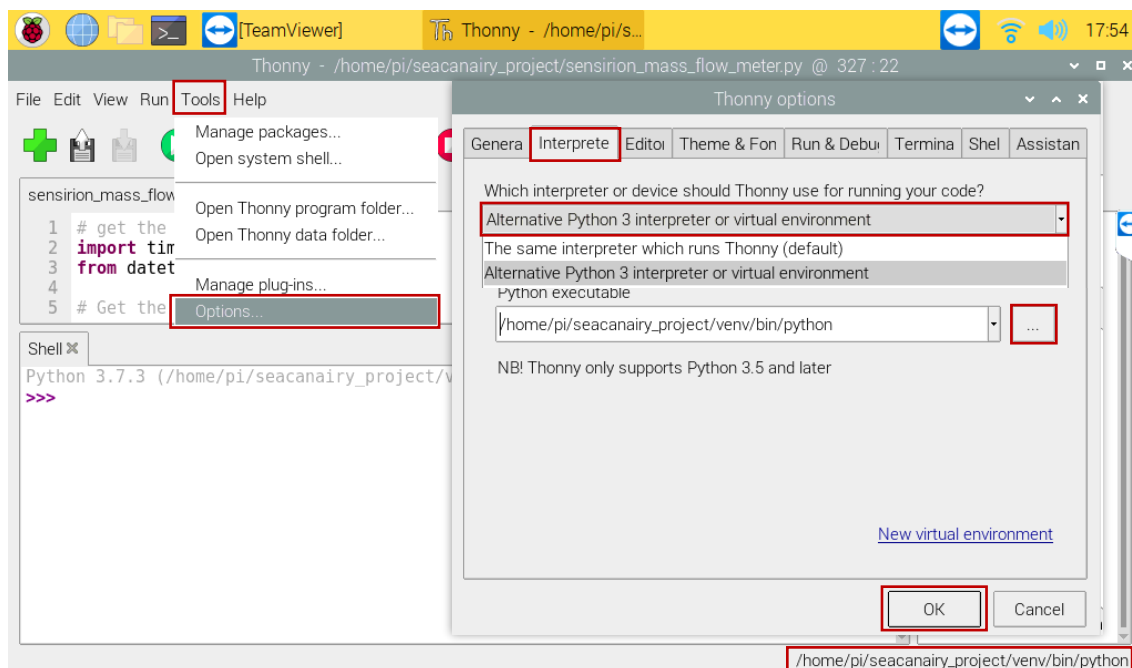


Figure 58 Virtual environment in Thonny Python IDE

Source: own work, a screenshot of the central computer screen, taken through TeamViewer

2.3 Install Python libraries on the Raspberry Pi

All libraries used in this project have been installed using `pip3` in the virtual environment. 1.5 on page 84.

Table 17 is a list of `pip3` functions necessary to install Python libraries. Before proceeding to any `pip3` execution, be sure to have activated the virtual environment as explained in point 2.2.2 on page 87. Necessary libraries for the Seacanairy are listed in 1.5 on page 84.

Table 17 pip3 function list

Source: own work, online documentation [16]

| Purpose | Function |
|--|------------------------------------|
| Show help | <code>pip3 --help</code> |
| Install a library (e.g., <code>smbus2</code>) | <code>pip3 install smbus2</code> |
| List all installed libraries | <code>pip3 list</code> |
| Remove a library (e.g., <code>smbus2</code>) | <code>pip3 uninstall smbus2</code> |

2.4 Testing code on the Raspberry Pi

As explained in 1 on page 80, it is best to write the code on your personal computer using PyCharm rather than working on the built-in Thonny Python IDE on the tiny Raspberry Pi's screen. However, sensors and devices are connected to the Raspberry Pi and not to our computer. Therefore, regular code transfer from the PC to the Raspberry Pi for testing is necessary. Different methods are available: copy/pasting the code as text using TeamViewer, file transfer using TeamViewer or python execution in the console.

2.4.1 Copy-pasting in Thonny Python IDE

Connect to the Raspberry Pi via TeamViewer as explained in point 1.6 on page 85, and open Thonny Python IDE as explained in point 2.2.3 on page 88. Next, select the whole code on your computer using `ctrl+A` and copy it to the clipboard `ctrl+c`. Next, go into TeamViewer, open Thonny Python IDE and paste the code in the corresponding file `ctrl+v`. To execute the code, press the green arrow on top of the window. Repeat this manipulation each time you need to test your code.

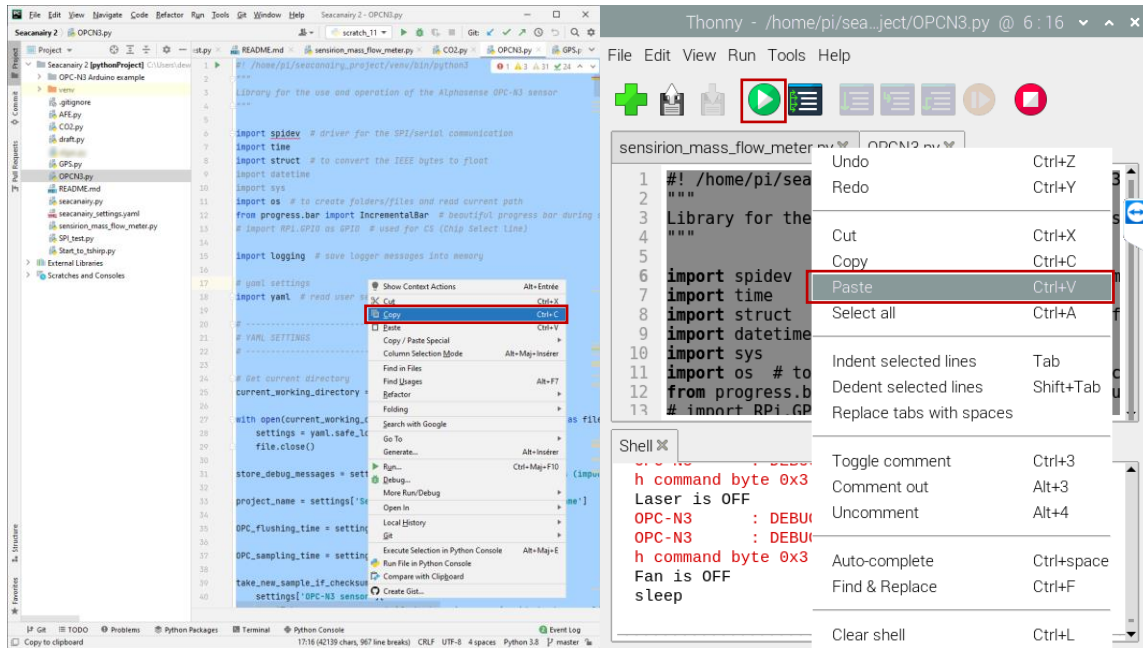


Figure 59 Copy-pasting code from PC to Thonny Python IDE

Source: own work, screenshot of the central computer screen, taken through TeamViewer

2.4.2 TeamViewer File Transfer and python3 in console

Open TeamViewer (as explained in point 1.6 on page 85) and open file transfer (as explained at point 1.7 on page 86). On the left side, browse through the maps where your code is located on your PC (refer to the PyCharm project creation in point 1.2 on page 81). On the right side, browse the place where you want to store the file on the Raspberry Pi (likely in the folder created before at point 2.2.1 on page 87). Finally, press on send. Once the file is stored on the Raspberry Pi, execute it using Python 3 in the console. Activate the virtual environment (see 2.2.2 on page 87). Then, write the following line, adapting `seacanairy.py` with the name of the code file you wish to execute (see Figure 60). Refer to point 3 (Console tip and tricks) for quicker console use.

```
python3 seacanairy.py
```

```

pi@raspberrypi: ~/seacanairy_project
File Edit Tabs Help
pi@raspberrypi:~ $ cd seacanairy_project
pi@raspberrypi:~/seacanairy_project $ source venv/bin/activate
(venv) pi@raspberrypi:~/seacanairy_project $ python3 seacanairy.py
##### TIME #####
Current date and time is: 2021-05-09 19:51:41.588993
If time is not correct, execute:
    sudo hwclock -s          apply RTC time to the system
    timedatectl             show current time status
##### FILES #####
SEACANAIRY : INFO    '/home/pi/seacanairy_project/long_term_test_29-04/long_te
rm_test_29-04-data.csv' already exist, appending data to this file
##### STARTING SENSORS #####
SEACANAIRY : INFO    Starting of Seacanairy on the 09/05/2021 at 19:51:42
CO2 sensor : INFO    Internal measuring time interval is 60 seconds
OPC-N3     : INFO    Fan speed is set on 100 (0 = the slowest, 100 = the faste
st)
Synchronize CO2 sensor with Seacanairy sampling period
Waiting for sensor sampling | ██████████ | 4/10 seconds
    
```

Figure 60 Testing code using file transfer and console

Source: own work, screenshot of the central computer scree, taken through TeamViewer

3 Console tip and tricks

The following table indicates some tip and tricks while working with the console. They make life easier and can save much time.

Table 18 Tip and tricks console

Source: own work

| Action | Description |
|------------------|---|
| Keyboard ↑ and ↓ | Move through the execution history (even if the system has been stopped/closed) |
| ctrl+c | Kill an instance (force to stop) |
| ctrl+x | Exit a nano screen |
| && | Concatenate different lines and actions in one line |
| | Example (in one line): cd seacanairy_project && source venv/bin/activate && python3 seacanairy.py |

4 Raspberry Pi password

Table 19 Raspberry Pi username and password

Source: own work

| | |
|----------|-------------|
| Username | pi |
| Password | raspberrypi |

Chapter 4

Software of the Seacanairy

The previous chapter described the environment in which the software was developed. In order to manage from the Seacanairy central computer the sensors, the data, operate the pump, start measurements at regular intervals, and store all the data in a single database, the software has been written in Python. This chapter deals with the overall structure of Python code files, how the different parts of the software work together and the other processes that take place during the operation of the Seacanairy. An in-depth study of the software for each sensor as well as their electrical connections is to be found in Chapter 1.

1 Overall Seacanairy software structure

The first problem we had to face during the realization of the Seacanairy is the good operation of the communications with the sensors. There is no standard allowing the easy plug and play connection of a sensor to a central unit and its immediate correct functioning. Therefore, each sensor required an in-depth study of the manufacturer's documentation, the communication protocol used, as well as hours of trial and error for the software to function correctly. The algorithms of each sensor have different functions, including `get_data()`, which automatically performs all the necessary operations to take a measurement and return the data. When each sensor is working individually, we wrote the final code for the Seacanairy, which starts the pump, execute the `get_data()` functions of each sensor, and stores the data in a single database. Figure 62 is a drawing of the global software structure. Each round indicates a different Python code file. Note that '.py' is the extension for Python files. Figure 62 is an illustration of importing the algorithms of the sensors and performing the functions to obtain the data.

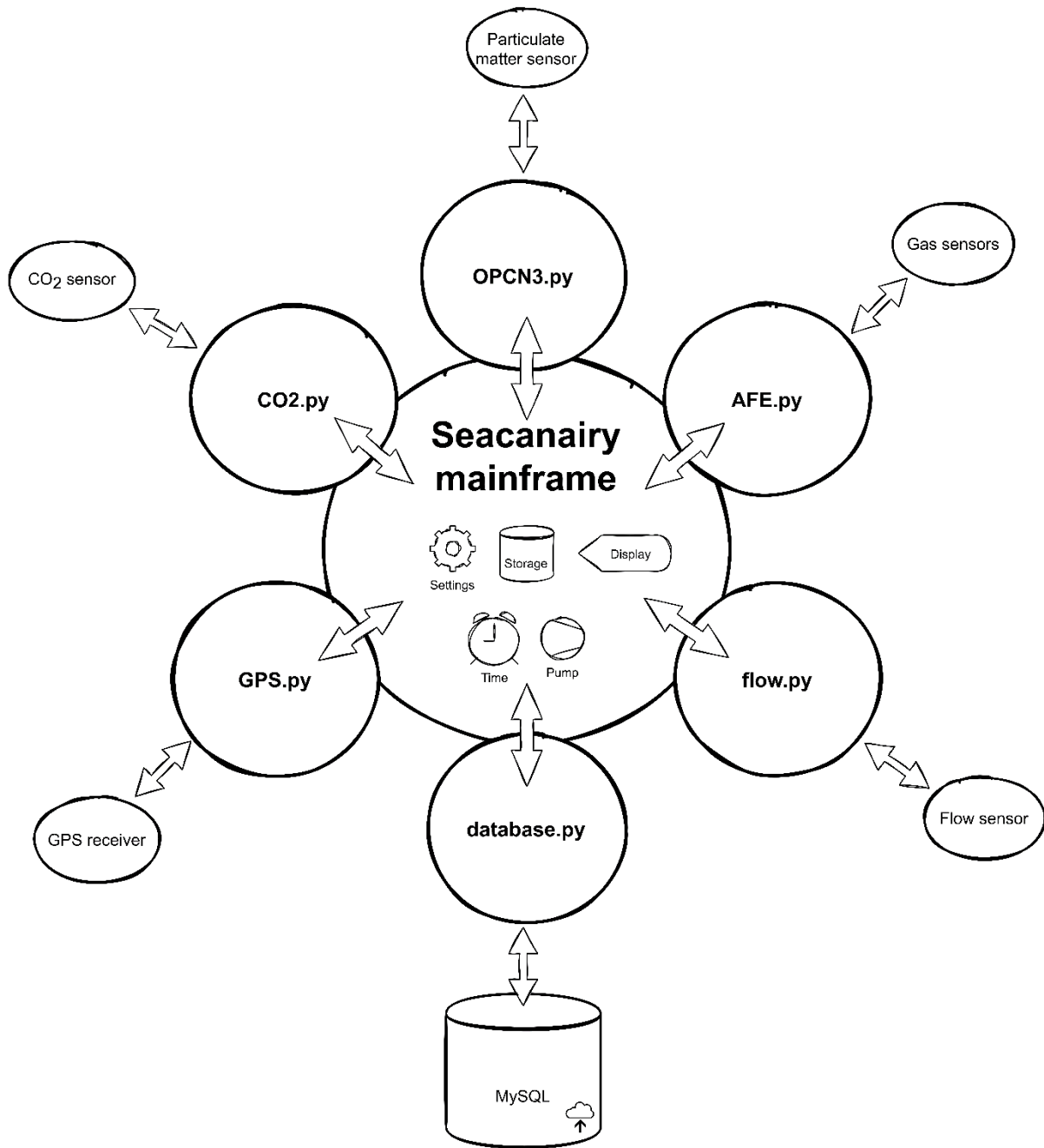


Figure 61 Seacanairy software structure

Source: own work, using draw.io


```
# Import Python files
import CO2
import OPCN3
import AFE
import GPS
import flow

# Use the functions from the Python files imported
CO2.get_data()
OPCN3.get_data(3, 5) # flushing_time=3, sampling_time=5
AFE.get_data()
GPS.get_position()
flow.get_data()
```

Figure 62 Importation in Python example

Source: own work, using PyCharm IDE

This working method is advantageous on the following points. The first advantage is the size of the file. Instead of having just one big file with thousands of rows, variables, and functions, we separate and structure our files for each sensor. Secondly, it makes it easier for anyone to copy our code and use it in another project. This is because all the components for a sensor are kept in a separate file. Finally, when we work on the software, we do not confuse functions and variables between the different sensors, many of them having the same names.

All of the code files we wrote follow the same structure. Indeed, several lines of code, such as obtaining settings and saving error messages, are common to all files. Figure 63 is a flowchart showing the general python script layout.

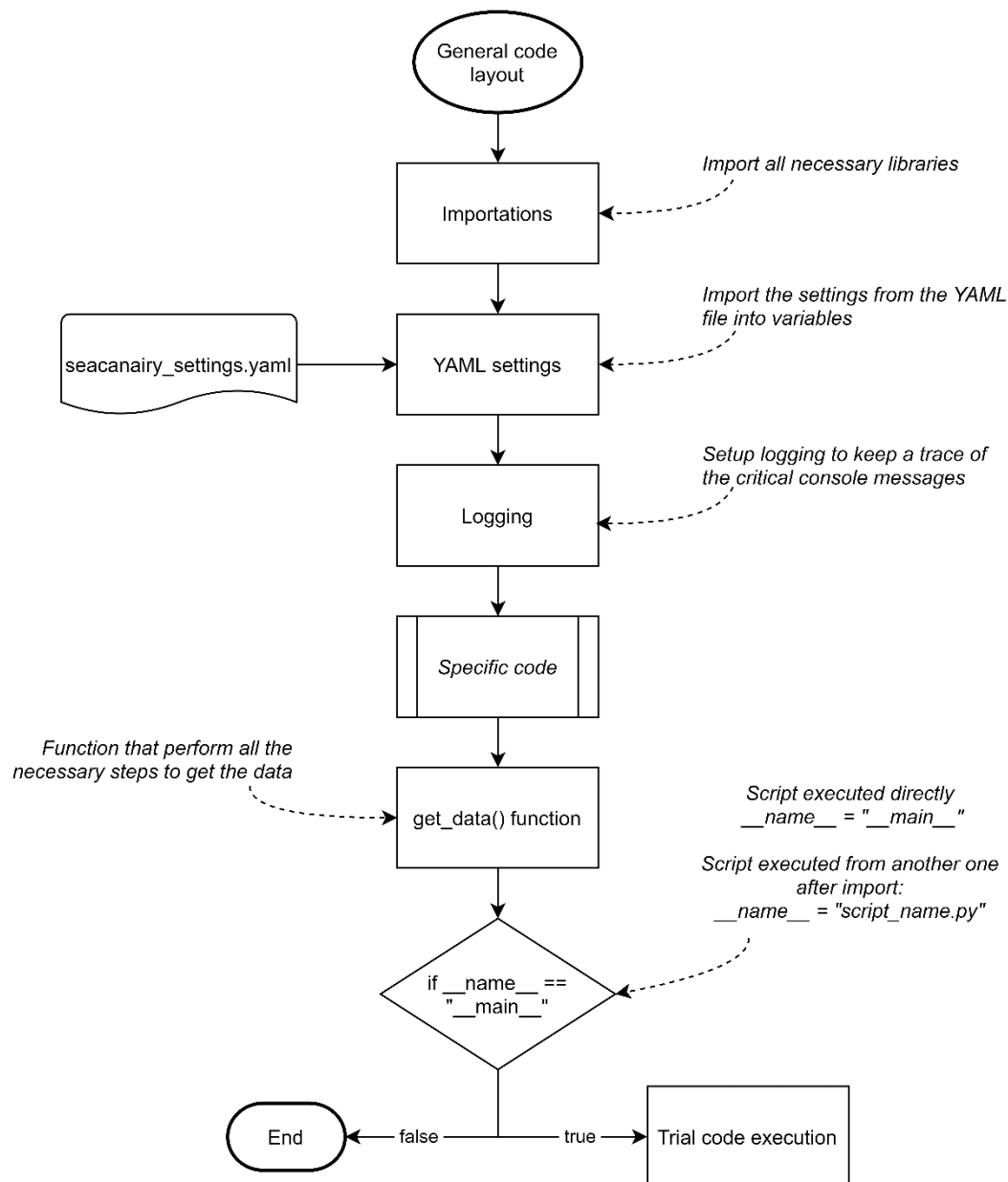


Figure 63 General Python script layout

Source: own work, using draw.io

2 Information display and logging functions

Logging is a crucial step when making computer code. This involves sorting and storing all the messages generated by our software in different levels, at least to the most important (debugging, information, warning, error and critical). In addition to being displayed on the screen, messages are stored in a separate text file. We then have access to a kind of logbook that keeps track of everything that has happened, for better or for worse. In addition, the Seacanairy is designed to operate autonomously for long periods of time. This allows us to check for any problems that may have arisen during this period of absence.

Depending on how the software is run, the messages displayed on the screen will be different. For example, running the entire Seacairy software will only show the most important messages. On the other hand, running the software of a particular sensor will show all available messages. Concretely, this makes it possible to have access to all the messages when working on the improvement of a particular sensor but to keep only the most important messages when the Seacairy is fully operational. That way, direct execution of a sensor script (i.e., running `CO2.py` directly) will display the messages. As the configuration of this module is a bit complex, it is accompanied by a flowchart in Figure 64 explaining the procedure. Note that a console handler is required to show the messages on the screen and store them into the log file. If the Seacairy script activates a handler and the script of a sensor also does, then the messages will be displayed twice on the screen. This is the reason why the handler is only activated when executing the script of a sensor directly (left part in Figure 64).

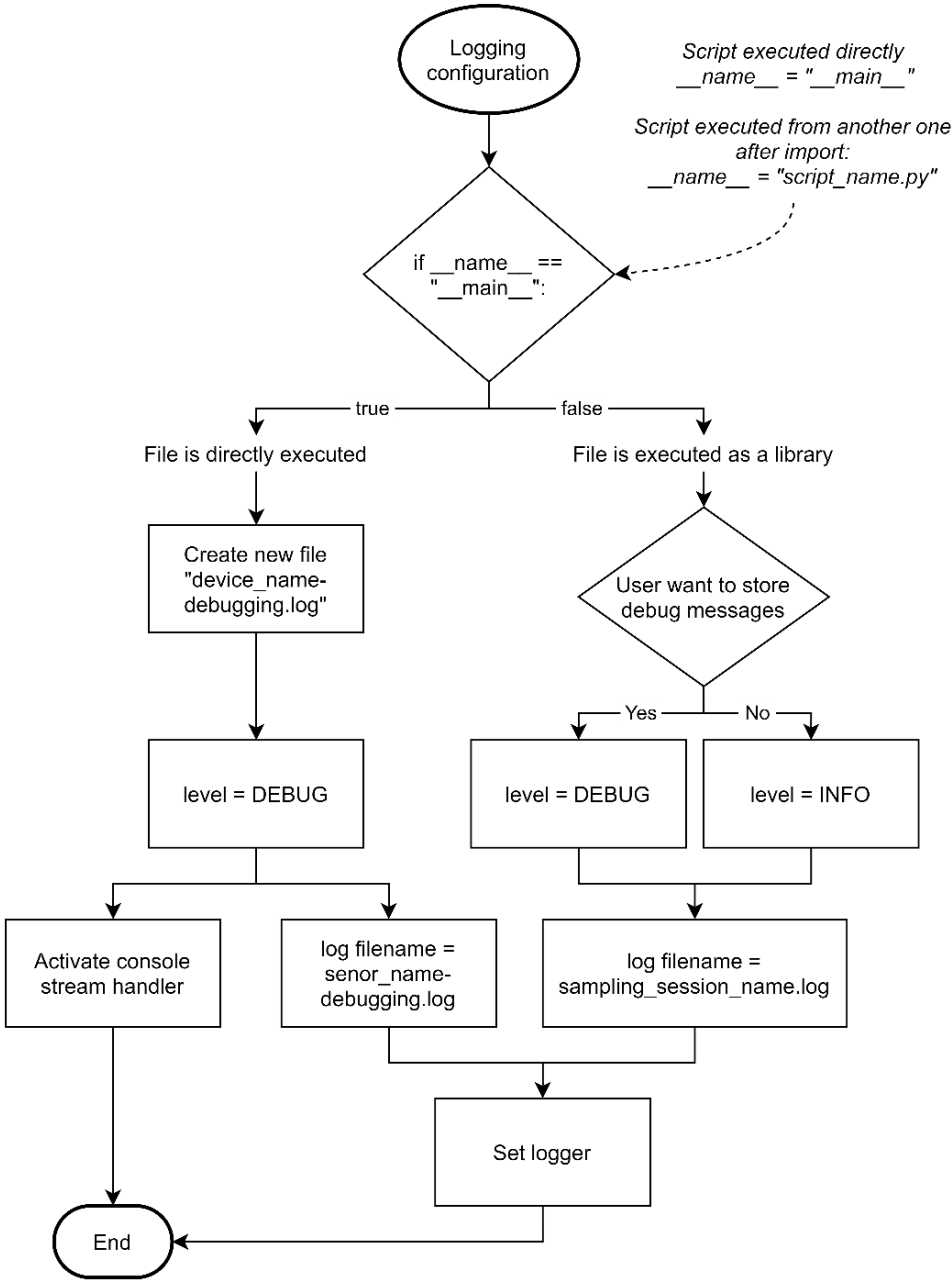


Figure 64 Logging flowchart

Source: on work, using draw.io

3 Settings page

3.1 Choice of file format

All the Python scripts depend on a unique settings file that allows the user to change a few sensors, logging, and sampling session settings without changing the source code. Two file types are available for that purpose: JSON and YAML. JSON (JavaScript Object Notation) is a derivative format of JavaScript. Working with indentation, brackets, colon, and quotations marks, almost every data type is available. Either readable by a machine or by a human, JSON

depend a lot on the syntax and do not allow any comments. YAML (Yet Another Markup Language) aims to be as easy as Python and rely only on indentation, indents, and colons. It also allows writing comments in the file after a number sign (#) to give more information about a setting. That makes YAML files clearer for people who know little about coding. Figure 65 is a comparison between JSON and YAML files for the same content.

JSON example

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```

YAML example

```
menu:
  id: file
  value: File
  popup:
    menuitem:
      - value: New
        onclick: CreateNewDoc()
      - value: Open
        onclick: OpenDoc()
      - value: Close
        onclick: CloseDoc()
```

Figure 65 Visual comparison between JSON and YAML

Source: adapted from Wikipedia [17]

3.2 Available settings

The `seacanairy_settings.yaml` file is a text file containing the various settings of the Seacanairy. A copy of this file can be found in the files joined to this paper (see Annexe 1 on page 113). To avoid any crash in the software, do not alter the structure of the file. Hereafter is a list of all the available settings.

General:

- Activate database/sensor: activate or deactivate a functionality.

Seacanairy:

- Sampling session name: name of the measurement session. This name will be assigned to the file containing the data, the logging file, and the database table.
- Sampling period: period of time between each measurement.
- Air pump minimum running time: minimum amount of time the air pump runs per loop. This ensures sufficient flushing of the air inside the piping. Gas sensor measurements starts after this amount of time.
- Store debug messages: store all messages in the logging file, or store only error messages.

MySQL database:

- Host: the URL leading to the server. Can also be an intranet IP address.
- Database name: name of the database
- User and password: different users can have access to a database with different authorizations.

CO₂ sensor:

- Automatic sampling frequency: number of measurements that the CO₂ sensor must perform per sampling period (setting defined previously) (see point 1.5 on page 14).
- Number of reading attempts: number of times the software tries to get the measurements if the checksum is wrong.

OPC-N3 sensor:

- Flushing time: period of time between the start of the laser and the fan, and the start of the measuring.
- Sampling time: amount of time laser and fan are running and taking sample.

- Fan speed: value between 0 and 100.
- Take a new measurement if the checksum is wrong: avoids too short measurement periods (see point 2.4.2 in Chapter 1 on page 29).

AFE Board:

- Noise reduction - number of reading averaged: number of successive measurements of gas concentrations to calculate the average in order to reduce the noise of the measurements (see point 3.3 on page 45).

4 MySQL Database

In parallel to an Excel file stored in the Raspberry Pi, the measurements are automatically stored in a MySQL database, hosted in the cloud or locally. Connection information must be specified in the Seacanairy settings file (see point 3.2 on page 99, as well as Annexe 11 on page 190). The same information can be used to link an Excel file on a standalone computer to retrieve the data from MySQL for remote data monitoring.

The flowchart in Figure 66 illustrates the connection process, followed by Figure 65, which shows how the systems update the database with new data, either coming from a new measurement or from older data not yet uploaded (i.e. due to an internet connection lost). Figure 68 shows how the data are shown in MySQL Workbench. They can also be downloaded directly to Microsoft Excel using a MySQL connector. That way, real-time graphs can be displayed. Graphs of the Seacanairy can be found in Annexe 14 on page 195.

The server used is a free service provided by remotemysql.com. Any other MySQL database host is possible as long as the server is remotely available, which is not the case with all free offers. It is also possible to host the MySQL server on a personal machine connected to the same router as the Seacanairy. In that case, the 'host' in the settings is the IP address of that machine.

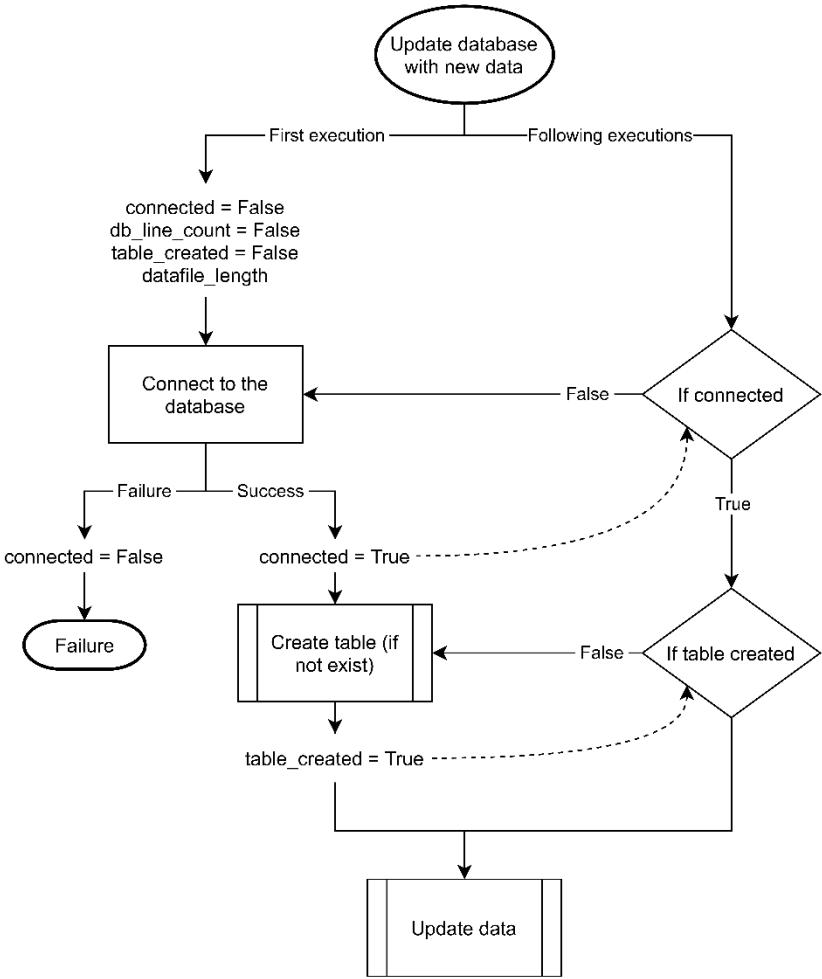


Figure 66 MySQL connection process flowchart

Source: own work, using draw.io

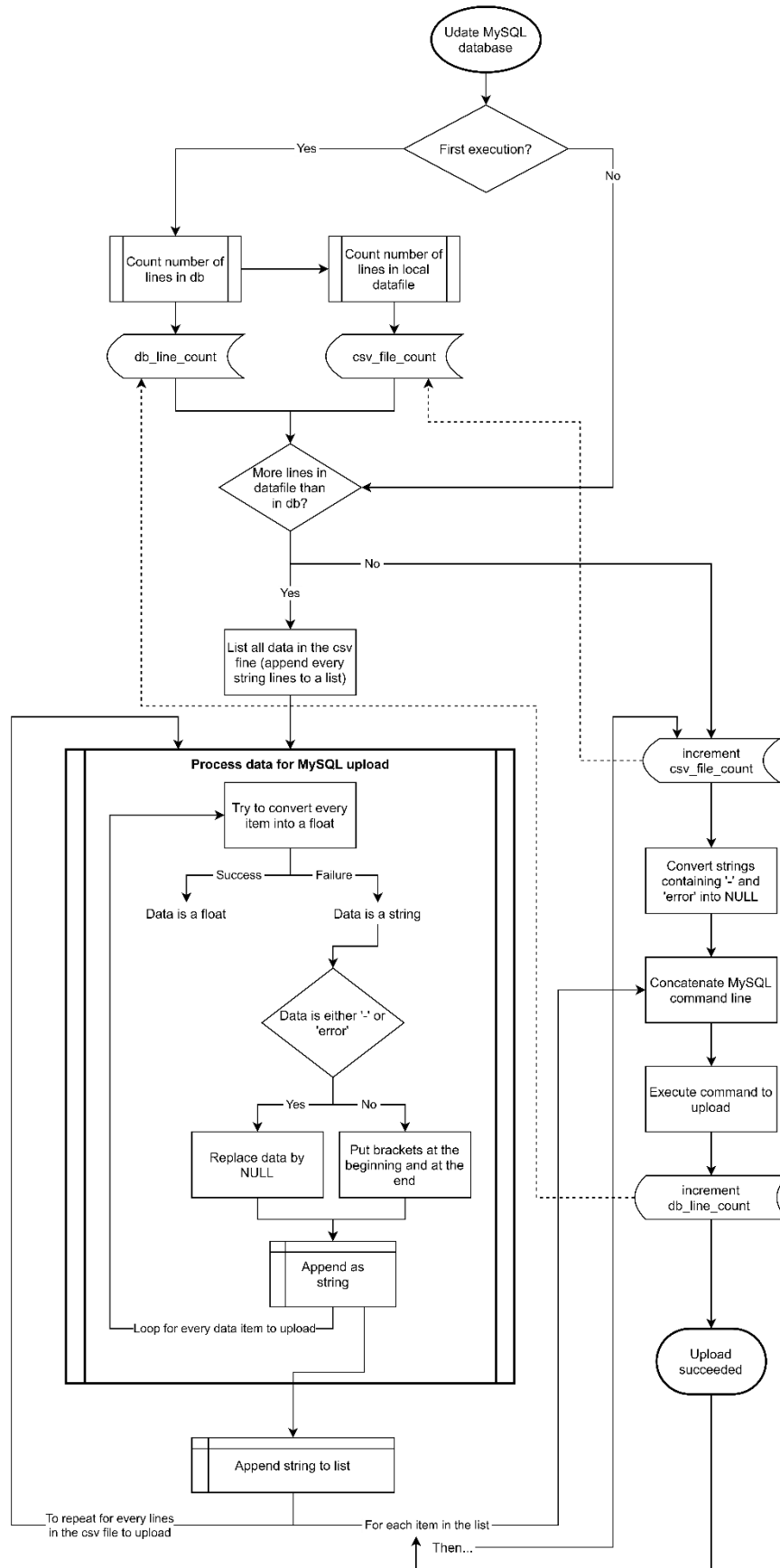


Figure 67 Database software flowchart

Source: own work, using draw.io

The screenshot shows the MySQL Workbench interface. The main window displays a query result grid for the table 'terrasse_bousval_24_08'. The query executed is 'SELECT * FROM 4tgGwNUHei.terrasse_bousval_24_08;'. The result grid contains 11 columns: Date_Time, PM_1, PM_2_5, PM_10, Temperature_OPC, Relative_Humidity_OPC, sampling_time_OPC, sample_flow_rate_OPC, bin_0, bin_1, bin_2, bin_3, bin_4, bin_5, bin_6, bin_7, bin_8, bin_9, and bin_10. The data shows various environmental measurements over time. The bottom panel shows the Action Output with three rows of query execution logs, indicating that the query was executed three times, each returning 85 rows.

| Date_Time | PM_1 | PM_2_5 | PM_10 | Temperature_OPC | Relative_Humidity_OPC | sampling_time_OPC | sample_flow_rate_OPC | bin_0 | bin_1 | bin_2 | bin_3 | bin_4 | bin_5 | bin_6 | bin_7 | bin_8 | bin_9 | bin_10 |
|---------------------|------|--------|-------|-----------------|-----------------------|-------------------|----------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|--------|
| 2021-08-24 17:08:00 | 0.96 | 1.04 | 1.04 | 31.27 | 33.87 | 1 | 5.31 | 62 | 7 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:09:00 | 1.45 | 2.42 | 2.78 | 31.27 | 35.36 | 1 | 5.31 | 67 | 17 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:10:00 | 1.17 | 1.3 | 1.3 | 31.28 | 35.42 | 1 | 5.31 | 62 | 13 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:11:00 | 1.11 | 1.37 | 1.38 | 31.28 | 35.39 | 1 | 5.4 | 57 | 10 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:13:17 | 1.06 | 1.54 | 1.59 | 31.31 | 33.92 | 1 | 5.8 | 61 | 7 | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:14:18 | 1.19 | 1.26 | 1.26 | 31.32 | 35.07 | 1 | 5.12 | 61 | 17 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:15:18 | 1.17 | 1.27 | 1.27 | 31.32 | 35.14 | 1 | 5.38 | 59 | 17 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:16:18 | 0.84 | 2.04 | 3.33 | 31.37 | 35.06 | 1 | 5.57 | 55 | 11 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:31:40 | 0.85 | 1.73 | 13.28 | 31.51 | 33.13 | 1 | 5.8 | 59 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 2021-08-24 17:32:40 | 0.98 | 1.09 | 1.09 | 31.49 | 34.65 | 1 | 5.58 | 58 | 9 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:41:53 | 0.75 | 1.08 | 1.12 | 31.51 | 34.6 | 1 | 7.02 | 47 | 16 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:42:53 | 0.91 | 2.65 | 4.24 | 31.52 | 35.22 | 1 | 6.41 | 57 | 14 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:43:53 | 0.74 | 0.81 | 0.81 | 31.53 | 35.14 | 1 | 6.24 | 46 | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:44:53 | 0.89 | 1.08 | 1.08 | 31.58 | 35.03 | 1 | 6.38 | 59 | 9 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:45:53 | 0.93 | 0.98 | 0.98 | 31.59 | 34.96 | 1 | 6.15 | 79 | 7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:46:53 | 1.08 | 1.29 | 1.3 | 31.61 | 34.92 | 1 | 6.09 | 71 | 11 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:47:53 | 1.01 | 1.79 | 2.1 | 31.62 | 34.86 | 1 | 6.13 | 72 | 9 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2021-08-24 17:48:53 | 1.02 | 1.11 | 1.12 | 31.65 | 34.69 | 1 | 5.93 | 55 | 17 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 68 Display of the data stored in the MySQL database

Source: own work, using MySQL Workbench

5 Global Seacnairy script

The Seacnairy script performs a series of tasks during its execution. Figure 69 is a flowchart of the different processes taking place. Since the software runs in a loop and takes measurements at regular intervals, there is no end to the diagram.

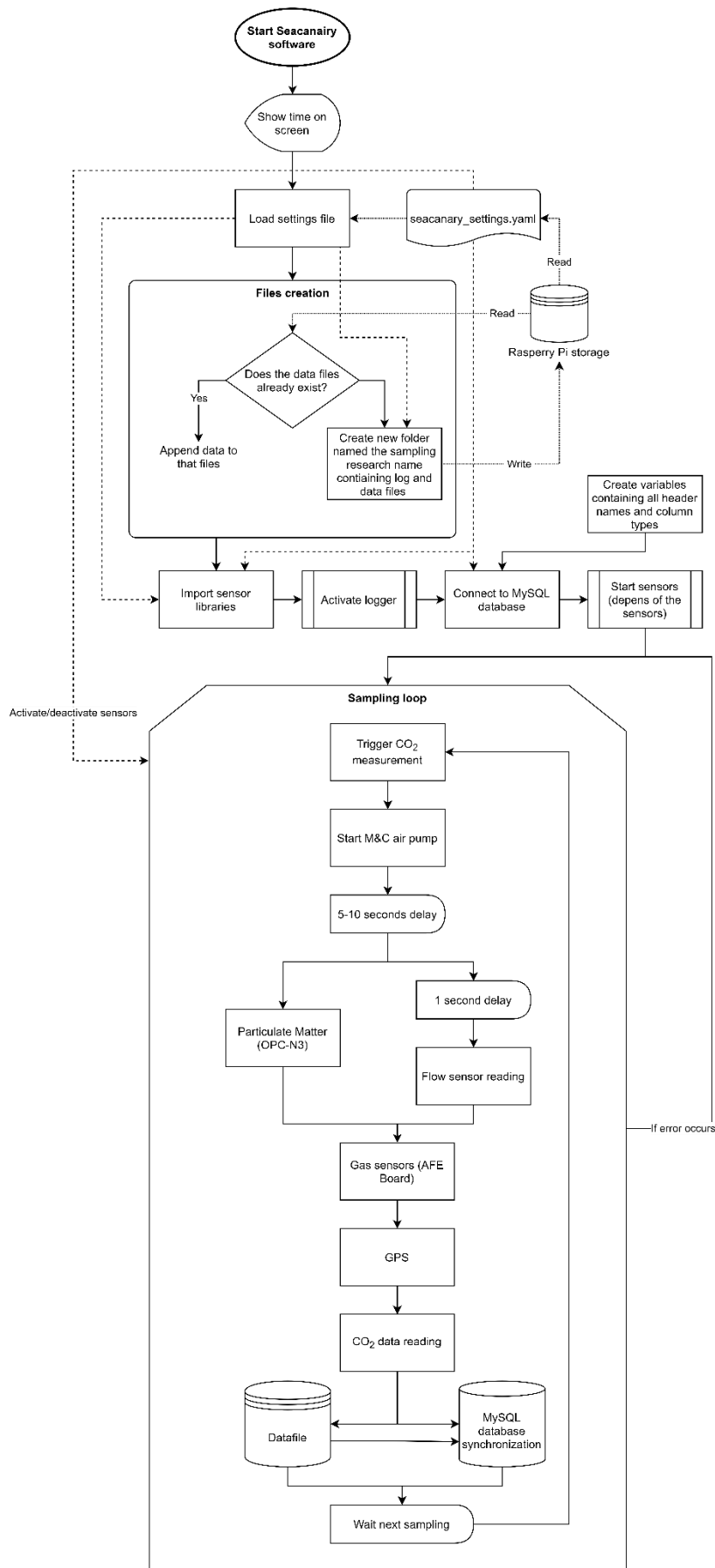


Figure 69 Flowchart showing Seacanairy software functioning

Source: own work, using draw.io

5.1 Manual operation through the touchscreen

On the Seacanairy screen (see Figure 70), several buttons allow the user to perform various operations, such as checking the system time, changing the Seacanairy parameters, or manually starting the Seacanairy.



Figure 70 Welcome screen of the Seacanairy (shown on the touchscreen)

Source: own work, using TeamViewer

5.2 Autostart at boot

A service, named `seacanairy.service`, has been created so that the system starts automatically after plugging in the power cable. When the user presses "Start Sampling", the background service stops to prevent the system from running twice at the same time. The following functions can be performed in order to change the behavior of the service.

Table 20 Functions to manage the Seacanairy service for autostart after boot

Source: own work

| Function | Operation |
|--|--|
| <code>sudo systemctl enable seacanairy.service</code> | Enable the service. Seacanairy will automatically start at next boot. |
| <code>sudo systemctl disable seacanairy.service</code> | Disable the service. Seacanairy will automatically start at next boot. |
| <code>sudo systemctl start seacanairy.service</code> | Start the service. In opposition to the shortcut on the touchscreen, Seacanairy will run in the background. |
| <code>sudo systemctl stop seacanairy.service</code> | Start the service. This is automatically performed while manually pressing « Start Sampling » on the screen. |
| <code>sudo systemctl restart seacanairy_service</code> | Restart the service. This should be executed to take into account settings changes. |

| | |
|--|--|
| <pre>sudo systemctl status seacanairy_service</pre> | <p>Get the status of the service. Shows last lines from the console. Indicate any restart due to any crash. Indicates the time the Seacanairy system has been running (see Figure 71).</p> |
|--|--|

```

seacanairy.service - Seacanairy sampling service
Loaded: loaded (/lib/systemd/system/seacanairy.service; enabled; vendor preset: enabled)
Active: active (running) since Tue 2021-08-24 19:54:52 CEST; 3s ago
Main PID: 18247 (python3)
Tasks: 1 (limit: 2062)
CGroup: /system.slice/seacanairy.service
└─18247 /home/pi/seacanairy_project/venv/bin/python3 /home/pi/seacanairy_project/seacanairy.py

Aug 24 19:54:52 raspberrypi systemd[1]: Started Seacanairy sampling service.
Aug 24 19:54:53 raspberrypi python3[18247]: MySQL      : INFO   Table (already) created (cuisine_bousval_24_08)
Aug 24 19:54:53 raspberrypi python3[18247]: SEACANAIRY  : INFO   '/home/pi/seacanairy_project/cuisine_bousval_24_08/cuisine_
Aug 24 19:54:53 raspberrypi python3[18247]: SEACANAIRY  : INFO   Appending data to this file
Aug 24 19:54:53 raspberrypi python3[18247]: SEACANAIRY  : INFO   Starting of Seacanairy on the 24/08/2021 at 19:54:53
Aug 24 19:54:53 raspberrypi python3[18247]: CO2 sensor  : INFO   Internal measuring time interval is 60 seconds
Aug 24 19:54:54 raspberrypi python3[18247]: OPC-N3     : INFO   Fan speed is set on 100 (0 = the slowest, 100 = the fastest

```

Figure 71 Seacanairy service status

Source : own work, using TeamViewer

6 Software files and folders

6.1 List of files

Figure 72 lists all the files that relate to the Seacanairy software. They are all stored in a dedicated folder named `seacanairy_project`.

The Seacanairy software has been written step by step, sensor by sensor. For each device, a separate Python sheet has been written that aims to manage communication, data interpretation and conversion properly. Each Python script is composed of functions that process variables, interpret data, communicate with the sensor, check the bytes... Finally, after hours of documentation readings, trial-and-errors, and online searches, the system succeeds in executing a `get_data()` function that aims to make all the necessary steps in good order to get the data from the sensor to the screen.

Each Python script generates a lot of messages printed on the screen. To store them for further analysis, a logging system sort all the messages according to their importance in a

dedicated log file. Message storing is an essential step in developing as we cannot wait for hours in front of the screen for error.

All the sensors are gathered via `seacairy.py`. This script starts the pump, call the `get_data()` functions of each sensor's Python script, and store all the data in a common comma-separated values file at regular interval. A settings file called `seacairy_settings.yaml` contains some sensor, sampling, and logging settings as well as sampling session names. In function of that last setting, `seacairy.py` will create a new folder with the sampling session name to store the data and the sensors' log.

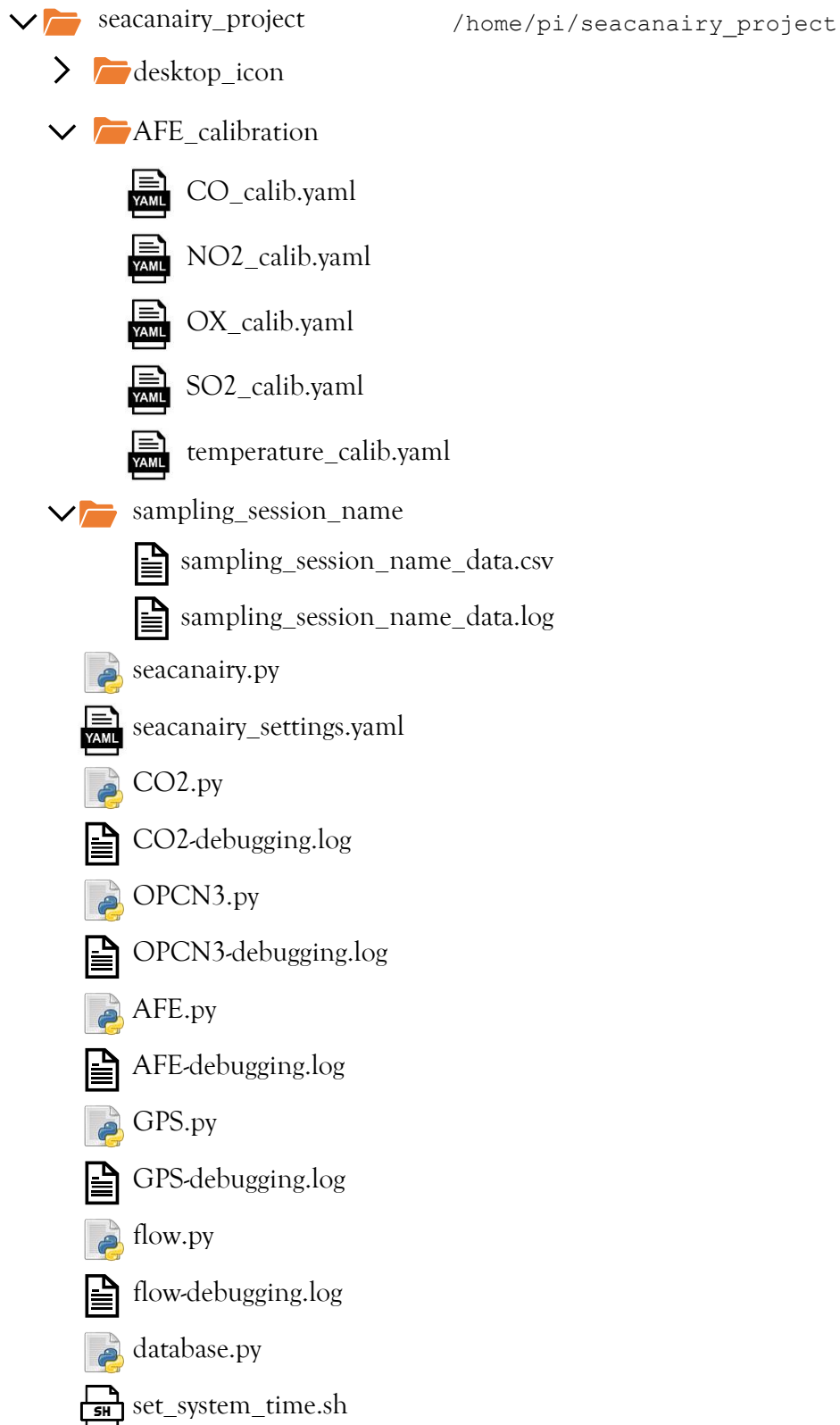


Figure 72 Files used by the Raspberry Pi for the proper execution of the software

Source: own work

Conclusion

This work proposed a design for a transportable, watertight, autonomous measuring instrument dedicated to measuring air quality onboard merchant ships that can be easily calibrated coupling calibration bottles to the tubing system. The proposed instrument measures sulfur oxides, nitrogen oxides, carbon oxides, ozone, particulate matter, temperature and humidity. Equipped with a GPS receiver, it also registers the vessel's position and behaviour, such as course or speed changes. Linked to an online database, it allows remote monitoring of the measurement taken through an internet connection.

The use of several different sensors ensures a wide measuring range of pollutants. First, a central computer connects the sensor to a Raspberry Pi via a printed circuit board. Then, Python software runs to performs all measurements at regular intervals and stores the data in a file. Next, a pump and a piping system bring the sampled air to the sensors. Using tubes makes it possible to connect calibration bottles to improve the sensor's accuracy. Also, an extension tube can be connected to measure the air from another place. Finally, all of the components are installed in a waterproof case so that all the necessary components become one single unit.

During the development of the Seacamairy, an extensive list of smaller and more significant problems have been encountered. Each of these problems had to be solved to have a properly working instrument. After troubleshooting, the measuring campaign performed with the device shows that the instrument is working.

Annexe 1

List of files

This document has been rendered with a compressed folder containing a series of files.
The list below mentions the files present.

- AFE_calibration
 - CO_calib.yaml
 - NO2_calib.yaml
 - OX_calib.yaml
 - SO2_calib.yaml
 - Temperature_calib.yaml
- AFE.py
- CO2.py
- database.py
- flow.py
- GPS.py
- OPCN3.py
- seacanairy.py
- seacanairy_settings.py
- set_system_time.sh

Annexe 2

Case panels dimensions

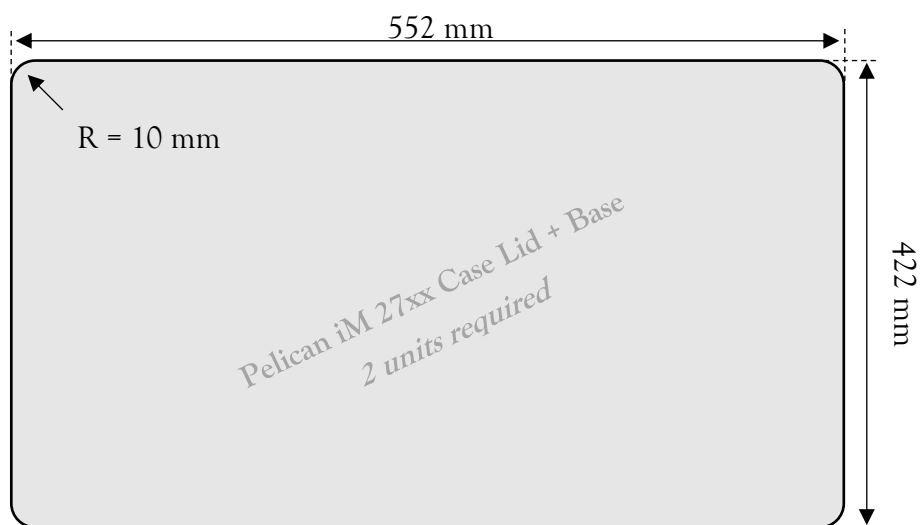


Figure 73 Lid and Base panel plan

Source: own measurements in the Pelican Case iM2720

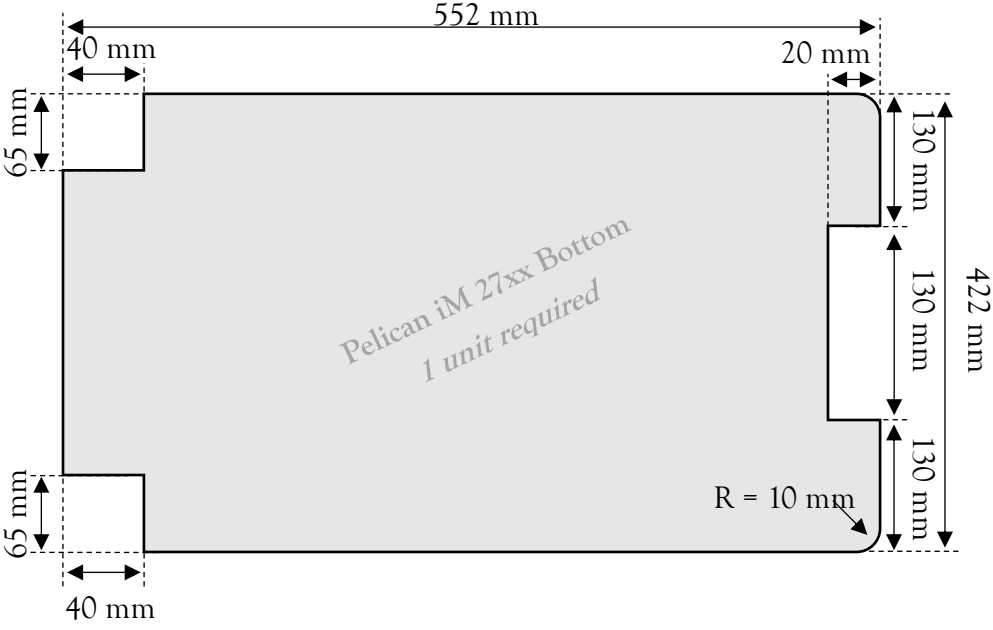


Figure 74 Bottom panel plan

Source : own measurements in the Pelican Case iM2720

Annexe 3

Schematic of the Seacanairy wiring

The following schematic is available in full scale format on next page.

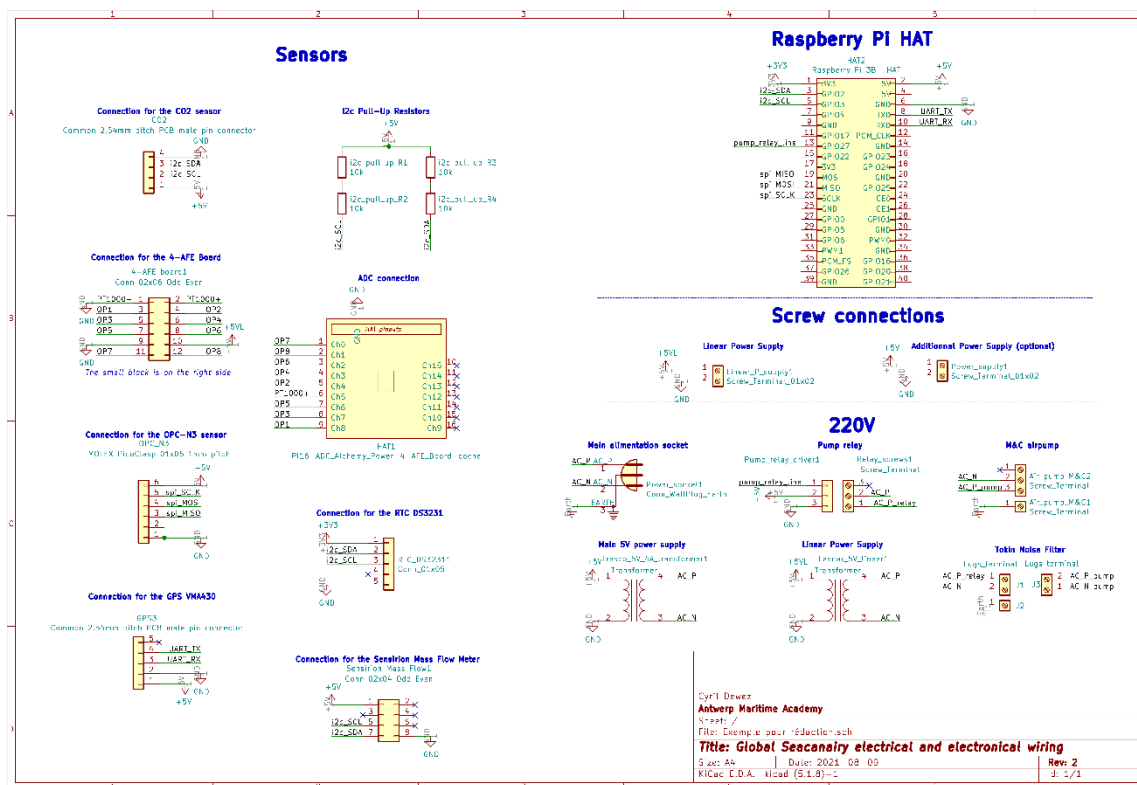
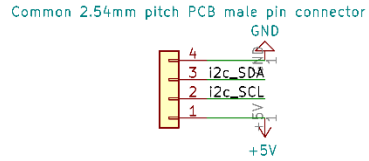


Figure 75 Seacanairy electronic and electric schematic

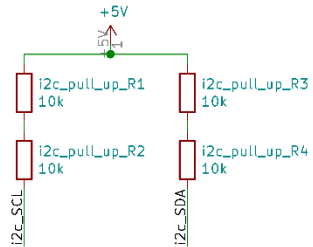
Source: own work using KiCad

Sensors

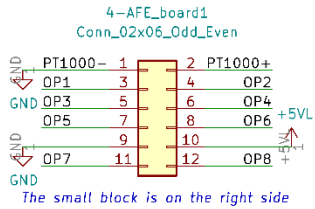
Connection for the CO2 sensor



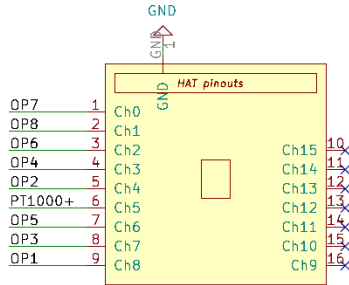
I2c Pull-Up Resistors



Connection for the 4-AFE Board

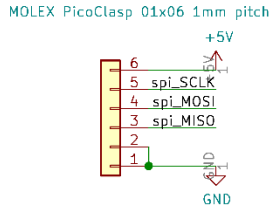


ADC connection

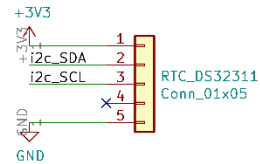


HAT1
PI16-ADC_Alchemy_Power-4-AFE_Board-cache

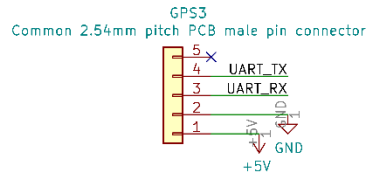
Connection for the OPC-N3 sensor



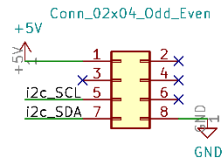
Connection for the RTC DS3231



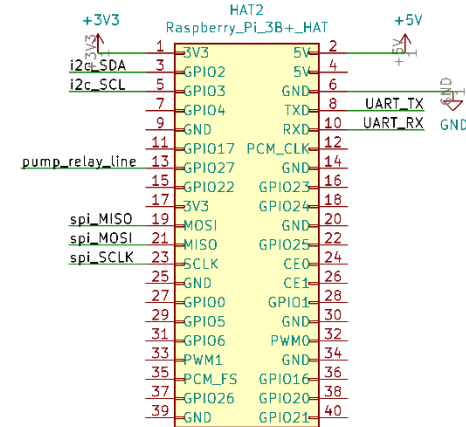
Connection for the GPS VMA430



Connection for the Sensirion Mass Flow Meter

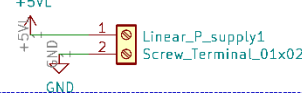


Raspberry Pi HAT

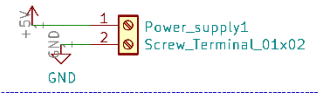


Screw connections

Linear Power Supply

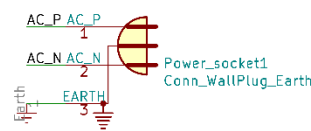


Additional Power Supply (optional)

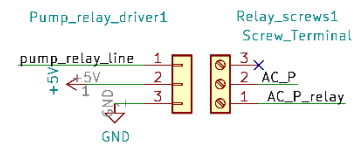


220V

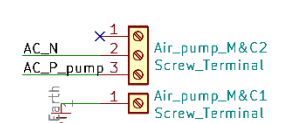
Main alimentation socket



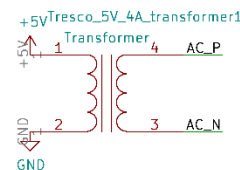
Pump relay



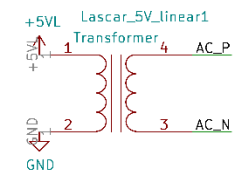
M&C airpump



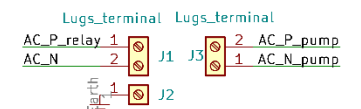
Main 5V power supply



Linear Power Supply



Tokin Noise Filter



Cyril Dewez
Antwerp Maritime Academy
Sheet: /
File: Exemple pour redaction.sch

Title: Global Seacnairy electrical and electronical wiring

Size: A4 Date: 2021-08-09
KiCad E.D.A. kicad (5.1.8)-1

Rev: 2
Id: 1/1

Annexe 4

Seacanairy PCB

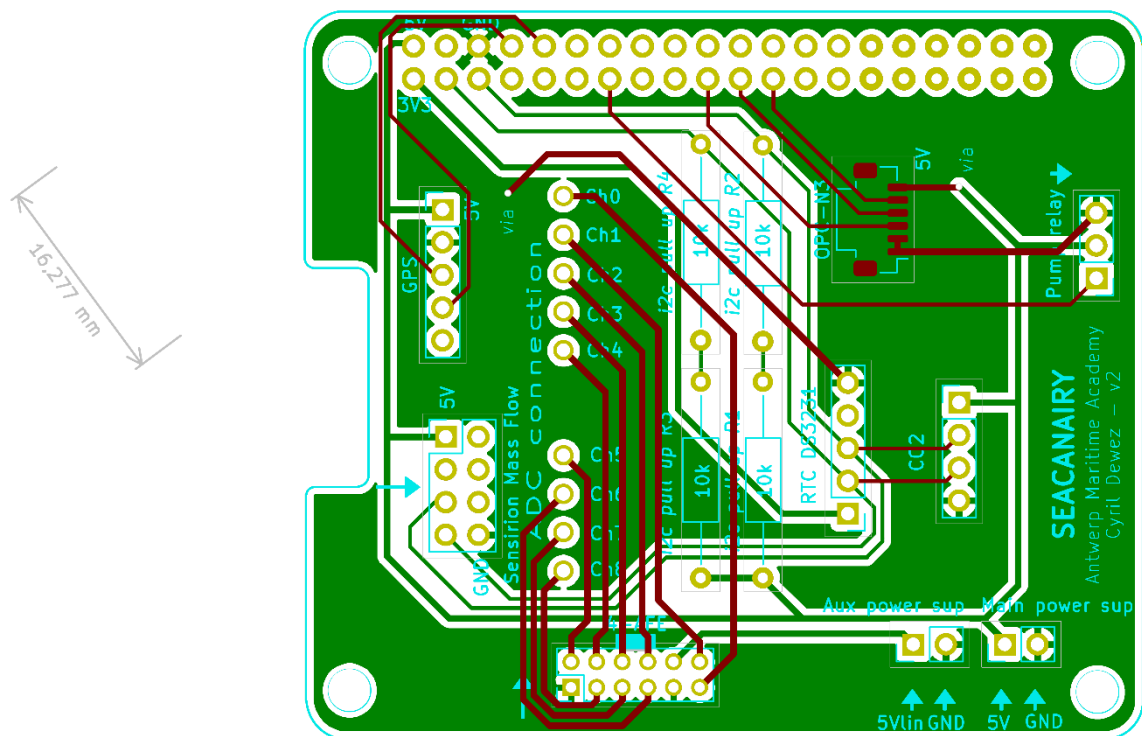


Figure 76 Seacanairy PCB version 2.0 (current version)

Source: own work, using KiCad

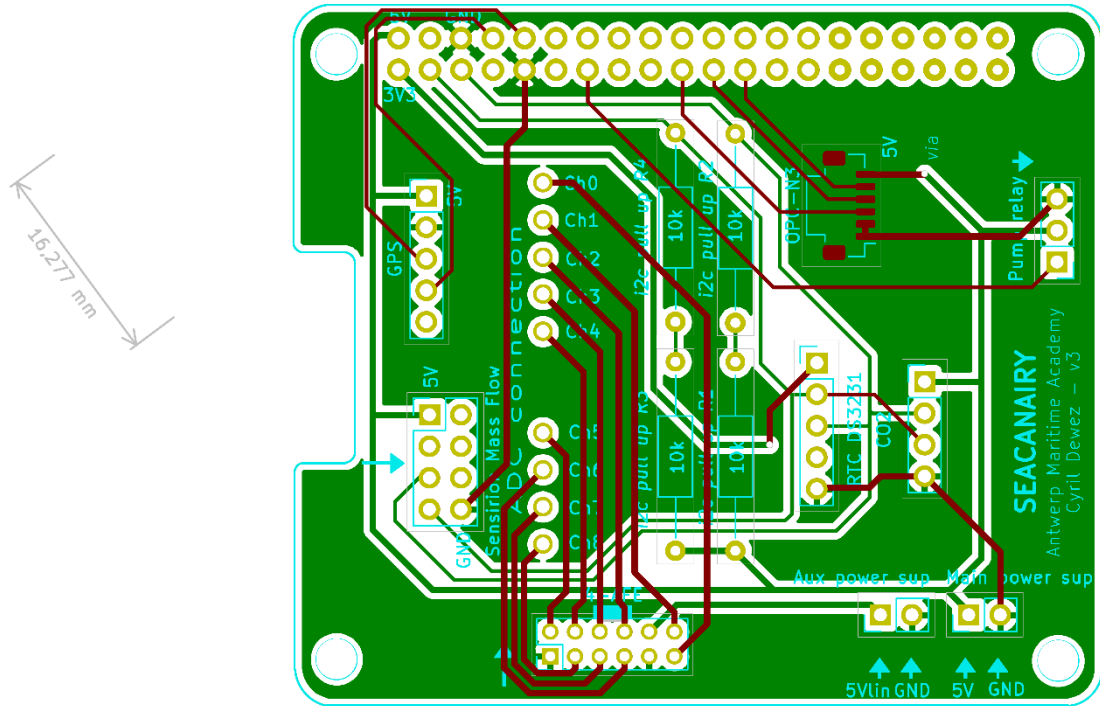


Figure 77 Seacanairy PCB version 3.0 (RTC DS3231 corrected)

Source: own work, using KiCad

Annexe 5

CO2.py

```
#!/home/pi/seacanairy_project/venv/bin/python3
"""
Library for the use of E+E Elektronik EE894 CO2 sensor via I2C
communication
"""
# -----
# USEFUL VARIABLES
# -----

# get the time
import time
from datetime import date, datetime

# Get the errors
import sys

# Create folders and files
import os

# smbus2 is the new smbus, allow more than 32 bits writing/reading
from smbus2 import SMBus, i2c_msg
# 'SMBus' is the general driver for i2c communication
# 'i2c_msg' allow to make i2c write followed by i2c read WITHOUT any STOP
byte (see sensor documentation)

# logging
import logging

# yaml settings
import yaml

# progress bar during sampling
from progress.bar import IncrementalBar

# I2C address of the CO2 device
CO2_address = 0x33 # i2c address by default, can be changed (see sensor
doc)

# emplacement variable
bus = SMBus(1) # make it easier to read/write to the sensor (bus.read or
bus.write...)

# -----
# YAML SETTINGS
# -----
# Get current directory
current_working_directory = str(os.getcwd())
```

```

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close() # close the file after use

store_debug_messages = settings['CO2 sensor']['Store debug messages
(important increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

measurement_delay = settings['CO2 sensor']['Amount of time required for the
sensor to take the measurement']

max_attempts = settings['CO2 sensor']['Number of reading attempts']

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward
# (check for errors fe)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s
%(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')

    logger = logging.getLogger('CO2 sensor') # name of the logger
    # all further logging must be called by logger.'level' and not
    logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/CO2-debugging.log"
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("CO2 Sensor DEBUG messages will be shown and stored in '" +
str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # The following HANDLER must be activated ONLY if you run this code
alone
    # Without the 'if __name__ == '__main__' condition, all the logging
messages are displayed 3 TIMES
    # (once for the handler in CO2.py, once for the handler in OPCN3.py,
and once for the handler in seacanairy.py)

    # define a Handler which writes INFO messages or higher to the
sys.stderr/display (= the console)
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')

```

```

# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger().addHandler(console)

else: # if this file is considered as a library (if you execute
seacanairy.py for example)
    # if the user asked to store all the messages in
'seacanairy_settings.yaml'
    if store_debug_messages:
        message_level = logging.DEBUG
    # if the user don't want to store everything
    else:
        message_level = logging.INFO
    # Create a file to store the log if it doesn't exist yet
    log_file = current_working_directory + "/" + project_name + "/" +
project_name + "-log.log"
    logger = set_logger(message_level, log_file)
    # no need to add a handler, because there is already one in
seacanairy.py

# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as ROOT and NOT 'CO2 sensor'

# -----

def loading_bar(name, delay):
    """
    Show a loading bar on the screen during a a certain amount of time
    Make the user understand the software is doing/waiting for something
    :param name: Text to be shown on the left of the loading bar (waiting,
sampling...)
    :param length: Amount of time the system is waiting (seconds)
    :return: nothing
    """
    bar = IncrementalBar(name, max=(2 * delay), suffix='%(elapsed)s/' +
str(delay) + ' seconds')
    for i in range(2 * delay):
        time.sleep(0.5)
        bar.next()
    bar.finish()
    return

def digest(buf):
    """
    Calculate the CRC8 checksum (based on the CO2 documentation example)
    :param buf: List of bytes to digest [bytes to digest]
    :return: checksum
    """
    # Translation of the C++ code given in the documentation
    crcVal = 0xff
    _from = 0 # the first item in a list is named 0
    _to = len(buf) # if there are two items in the list, then len() return
1 --> range(0, 1) = 2 loops

    for i in range(_from, _to):
        curVal = buf[i]

```

```

    for j in range(0, 8): # C++ stops when J is not < 8 --> same for
python in range
        if ((crcVal ^ curVal) & 0x80) != 0:
            crcVal = (crcVal << 1) ^ 0x31

        else:
            crcVal = (crcVal << 1)

            curVal = (curVal << 1) # this line is in the "for j" loop, not
in the "for i" loop

    checksum = crcVal & 0xff # keep only the 8 last bits

    return checksum

def check(checksum, data):
    """
    Check that the data transmitted are correct using the data and the
    given checksum
    :param checksum: Checksum given by the sensor (see sensor doc)
    :param List of bytes transmitted by the sensor before the checksum (see
    sensor doc)
    :return: True if the data are correct, False if not
    """
    calculation = digest(data)
    if calculation == checksum:
        logger.debug("CRC8 is correct, data are valid")
        return True
    else:
        logger.debug("CRC8 does not fit, data are wrong")
        logger.error("Checksum is wrong, sensor checksum: " + str(checksum)
+
                        ", seacanairy checksum: " + str(calculation) +
                        ", bytes returned:" + str(data) + str(checksum))
        if data[0] and data[1] == 0:
            logger.debug("Sensor returned 0 values, it is not ready,
waiting a little bit")
            print("Sensor not ready, waiting...", end='\r')
            time.sleep(3)
            return False

def status(print_information=True):
    """
    Read the status byte of the CO2 sensor
    !! It will trigger a new measurement if the previous one is older than
10 seconds
    :param: print_information: Optional: False to hide the messages
    :return: True if last measurement is OK, False if NOK
    """
    logger.debug("Reading sensor status")
    try:
        with SMBus(1) as bus:
            # reading = read_from_custom_memory(0x71, 1)
            reading = bus.read_byte_data(CO2_address, 0x71)
            # see documentation for the following decryption
            CO2_status = reading & 0b00001000
            temperature_status = reading & 0b00000010
            humidity_status = reading & 0b00000001
            if print_information: # if user/software indicate to print the
information

```

```
    if CO2_status == 0:
        logger.debug("CO2 status is OK")
    else:
        logger.warning("CO2 status is NOK")
    if temperature_status == 0:
        logger.debug("Temperature status is OK")
    else:
        logger.warning("Temperature is NOK")
    if humidity_status == 0:
        logger.debug("Humidity status is OK")
    else:
        logger.warning("Humidity status is NOK")
    if CO2_status or humidity_status != 0:
        # Only CO2_status and humidity_status, because for no known
reason temperature status is always NOK
        return False
    else:
        # Everything is OK
        return True
except:
    logger.critical("Status reading failure")
    return True # try to go ahead in all cases

def getRHT():
    """
    Read the last Temperature and Relative Humidity measured, process the
    bytes, check checksum, convert in °C and %RH
    :return: Dictionary with the following items {"RH", "temperature"}
    """
    logger.debug("Reading RH and Temperature from CO2 sensor")

    write = i2c_msg.write(CO2_address, [0xE0, 0x00]) # see documentation,
example for reading t° and RH
    read = i2c_msg.read(CO2_address, 6)

    attempts = 0 # trial counter for the checksum and the validity of the
data received
    reading_trials = 0 # trial counter for the i2c communication

    # In case there is a problem and it return nothing, return "error"
    data = {
        "relative humidity": "error",
        "temperature": "error"
    }

    # all the following code is in a loop so that if the checksum is wrong,
it start a new measurement
    while attempts <= max_attempts:

        while reading_trials <= max_attempts: # reading loop, will try
again if the i2c communication fails
            try: # SMBUS stop working in case of error, avoid the software
to crash in case of i2c error
                with SMBus(1) as bus:
                    bus.i2c_rdwr(write, read)
                break # break the loop if the try has not failed at the
previous line, jump to the process of data

            except: # what happens if the i2c fails
                if reading_trials == max_attempts:
                    logger.critical("i2c failure "
```

```

+ str(max_attempts) + "consecutive
times, skipping this RH and temperature reading")
    return data # indicate clearly that data are wrong

    logger.error("i2c failure (" + str(sys.exc_info())
                + "), trying again... (" + str(reading_trials
+ 1) + "/" + str(max_attempts) + ")")
    reading_trials += 1 # increment of reading_trials
    time.sleep(3) # if transmission fails, wait a bit to try
again (sensor is maybe busy)

    # process the data given by the sensor
    reading = list(read)
    # if the two checksums are correct...
    if check(reading[2], [reading[0], reading[1]]) and
check(reading[5], [reading[3], reading[4]]):
    # reading << 8 = shift bytes 8 times to the left, say
differently, add 8 times 0 on the right
    temperature = round(((reading[0] << 8) + reading[1]) / 100 -
273.15, 2)
    relative_humidity = ((reading[3] << 8) + reading[4]) / 100

    print("Temperature is:", temperature, "°C", end="")
    print("\t| Relative humidity is:", relative_humidity, "%RH")

    # Create a dictionary containing all the data
    data = {
        "relative humidity": relative_humidity,
        "temperature": temperature
    }

    return data

else: # if one or both checksums are not corrects
    if attempts == max_attempts:
        logger.error("Data were wrong "
                    + str(max_attempts) + " consecutive times,
skipping this RH and temperature reading")
        return data # indicate on the SD card that data are wrong

    else:
        attempts += 1
        logger.warning("Error in the data received (wrong
checksum), reading data again... ("
                    + str(attempts) + "/" + str(max_attempts) +
)")
        time.sleep(4) # avoid to close i2c communication

def getCO2P():
    """
    Read the last CO2 instant, CO2 average and pressure measurements,
    process the bytes, check checksum,
    convert in hPa and ppm
    :return: Dictionary containing the following items {"average",
"instant", "pressure"}
    """
    logger.debug('Reading of CO2 and pressure')

    write = i2c_msg.write(CO2_address, [0xE0, 0x27]) # see documentation,
reading of CO2 and pressure example
    read = i2c_msg.read(CO2_address, 9)

```



```
attempts = 0 # trial counter for the checksum and the validity of the
data received
reading_trials = 0 # trial counter for the i2c communication

# Create a dictionary containing the data, return "error" in case of
error
data = {
    "average": "error",
    "instant": "error",
    "pressure": "error"
}

# all the following code is in a loop so that if the checksum is wrong,
it start a new measurement
while attempts <= max_attempts:

    while reading_trials <= max_attempts: # reading loop, will try
again if the i2c communication fails
        try: # SMBUS stop working in case of error, avoid the software
to crash in case of i2c error
            with SMBus(1) as bus:
                bus.i2c_rdwr(write, read)
            break # break the loop if the try has not failed at the
previous line, jump to the process of data

        except: # what happens if the i2c fails
            if reading_trials == max_attempts:
                logger.critical("i2c failure "
                    + str(max_attempts) + " consecutive
times, skipping CO2 and pressure reading (" +
                    str(sys.exc_info()) + ")")
                return data # indicate clearly that the data are wrong

            logger.error("i2c failure, trying again... (" +
str(sys.exc_info()) + ")")
            reading_trials += 1 # increment of reading_trials
            print("Waiting 3 seconds...", end='\r')
            time.sleep(3) # if I2C comm fails, wait a little bit and
try again (sensor is maybe busy)

            # process the data given by the sensor
            reading = list(read)
            # if the two checksums are correct...
            if check(reading[2], [reading[0], reading[1]]) and
check(reading[5], [reading[3], reading[4]]) and check(
                reading[8], [reading[6], reading[7]]):
                pressure = (((reading[6] << 8) + reading[7]) / 10 # reading
<< 8 = shift bytes 8 times to the left
                print("Pressure is:", pressure, "mbar")

                CO2_average = (reading[0] << 8) + reading[1] # reading << 8 =
shift bytes 8 times to the left
                print("CO2 average is:", CO2_average, "ppm", end="")

                CO2_raw = (reading[3] << 8) + reading[4]
                print("\t| CO2 instant is:", CO2_raw, "ppm")

            data = {
                "average": CO2_average,
                "instant": CO2_raw,
                "pressure": pressure
```

```

    }

    return data

    else: # if one or both checksums are not corrects
        if attempts == max_attempts:
            logger.error("Error in the data received (wrong checksum),
skipping this CO2 and pressure reading")
            return data # indicate clearly that the data are wrong

        else:
            attempts += 1
            logger.warning("Error in the data received (wrong
checksum), reading data again... (" +
                           str(attempts) + "/" + str(max_attempts) +
                           ")")
            time.sleep(3) # avoid too close i2c communication

def get_data():
    """
    Get all the available data from the CO2 sensor (CO2 instant/average,
    pressure, temperature, humidity
    :return: Dictionary containing the following items
             {"pressure", "temperature", "CO2 average", "CO2 instant",
"relative humidity"}
    """
    # Read status byte
    # attempts = 1
    # while True:
    #     if status(True):
    #         break
    #     else:
    #         print("Waiting for data to be ready...", end='\r')
    #         time.sleep(2)
    #         attempts += 1
    #     if attempts >= 6:
    #         print("Sensor not ready, trying to read...", end='\r')
    #         break

    # Get CO2 and pressure
    data1 = getCO2P()
    # Get RH and temperature
    data2 = getRHT()
    # Append those two dictionary
    data1.update(data2)
    return data1

# -----
# Settings
# -----

def internal_timestamp(new_timestamp=None):
    """
    Read the internal sampling period of the CO2 sensor
    To change the value, write it between the brackets (in seconds)
    :param new_timestamp: None or empty to read, new value in seconds to
change it
    :return: Actual internal sampling period of the sensor
    """
    if new_timestamp is not None: # if user write something as input in

```

HZS

```
the brackets (arguments)
    if not 15 <= new_timestamp <= 3600:
        logger.warning("Sampling period should be a number between 15
and 3600 seconds (see sensor documentation)")
        to_write = new_timestamp * 10 # see sensor documentation
        msb_timestamp = (to_write & 0xFF00) >> 8
        lsb_timestamp = (to_write & 0xFF)
        reading = write_to_custom_memory(0x00, msb_timestamp,
lsb_timestamp)
    else: # if user doesn't write anything between the brackets
        reading = read_from_custom_memory(0x00, 2)

    if reading is not False: # read_from_custom_memory() returns False in
case of error...
        # ...Python crash if it tries to make calculations with a boolean
(True or False)
        measuring_time_interval = (reading[1] + reading[0] * 256) / 10
        if new_timestamp is None: # adapt the message in function of the
wishes of the user (here he want to read)
            logger.info("Internal measuring time interval is " +
str(int(measuring_time_interval)) + " seconds")
        else: # (here he want to write)
            logger.info(
                "Internal measuring time interval set successfully on " +
str(int(measuring_time_interval)) + " seconds")
            return measuring_time_interval
        else:
            logger.error("Failed to change the internal timestamp to " +
str(new_timestamp) + " seconds")

def trigger_measurement(force=False):
    """
    Request a new CO2, t°, pressure and RH measurement IF the previous one
is older than 10 seconds
    Force to avoid the previous 10 seconds condition
    Same function as 'status()'
    :param: force: True to apply the function two consecutive times to be
sure that the sensor is well
                    synchronized with the seacanairy
                    False to apply it once (during the main loop of the
Seacanairy for example)
    :return: True or False if status if OK or NOK
    """
    print("Triggering a new measurement...", end='\r')

    # The sensor will not take a new sample if the previous one is older
than 10 seconds
    sensor_status = status(False) # trigger new measurement

    if force: # if force is True
        if measurement_delay != 0: # if user/software want to wait for the
data to be ready
            loading_bar("Waiting for sensor sampling", measurement_delay)
# usually 10 seconds (see doc)
            # sensor documentation, let time to the sensor to perform the
measurement

            # That way, we ensure that the sensor will trigger a new
measurement RIGHT now
            sensor_status = status(False)
            loading_bar("Waiting for sensor sampling", measurement_delay)
```

```

# sensor documentation, let time to the sensor to perform the
measurement

    return sensor_status # same function as 'status()', but here we don't
    want to print the status on the screen

def read_internal_calibration(item):
    """
    Read the internal temp_calib of a particular sensor item
    :param item: indicate which internal temp_calib to read: 'relative
    humidity', 'temperature', 'pressure', 'CO2', 'all'
    :return: List containing the temp_calib settings [offset, gain,
    lower_limit, upper_limit]
    """
    if item == 'relative humidity':
        index = 0x01
        unit = "%RH"
        factor = 1 / 100
    elif item == 'temperature':
        index = 0x02
        unit = "Kelvin"
        factor = 1 / 100
    elif item == 'pressure':
        index = 0x03
        unit = "mbar"
        factor = 1 / 10
    elif item == 'CO2':
        index = 0x04
        unit = "ppm"
        factor = 1
    elif item == "all":
        for i in ['relative humidity', 'temperature', 'pressure', 'CO2']:
# iterate this function for each parameter
            read_internal_calibration(i)
            time.sleep(0.5) # avoid too close i2c communication
        return # exit the function once the iteration is finished
    else:
        raise TypeError("Argument of read_internal_calibration is wrong,
        must be: 'relative humidity', "
            "'temperature', 'pressure', 'CO2' or 'all'")

    reading = read_from_custom_memory(index, 8)

    if reading is False: # if read_from_custom_memory() function doesn't
    work, will return False...
        logger.error("Failed to read the internal temp_calib of the CO2
        sensor")
        return False # indicate error
    print(reading)
    offset = (reading[0] << 8 + reading[1]) * factor
    gain = (reading[2] << 8 + reading[3]) / 32768
    lower_limit = (reading[4] << 8 + reading[5]) # factor taken into
    account further
    upper_limit = (reading[6] << 8 + reading[7]) # factor taken into
    account further

    logger.info("Reading temp_calib for " + str(item) + ":")
    logger.info("\tOffset: " + str(offset) + " " + str(unit))
    logger.info("\tGain: " + str(gain))
    if lower_limit == 0xFFFF:
        logger.info("\tNo last lower limit adjustment")

```

```

        lower_limit = 0
    else:
        lower_limit += factor
        logger.info("\tLower limit: " + str(lower_limit) + " " + str(unit))
    if upper_limit == 0xFFFF:
        logger.info("\tNo last upper minute adjustment")
        upper_limit = 0
    else:
        upper_limit *= factor
        logger.info("\tUpper limit: " + str(upper_limit) + " " + str(unit))
    return [offset, gain, lower_limit, upper_limit]

def read_from_custom_memory(index, number_of_bytes):
    """
    Read bytes from specified custom memory address in the CO2 sensor
    internal memory
    :param index: index of the data to be read (see sensor doc)
    :param number_of_bytes: number of bytes to read (see sensor doc)
    :return: list[bytes] from right to left
    """
    logger.debug("Reading " + str(number_of_bytes) + " bytes from customer
memory at index " + str(hex(index)) + "...")
    write = i2c_msg.write(CO2_address, [0x71, 0x54, index]) # usual bytes
to send/write to initiate the reading
    attempts = 1
    read = [] # avoid return issue

    while attempts < 4:
        try:
            with SMBus(1) as bus:
                bus.i2c_rdwr(write)
            read = i2c_msg.read(CO2_address, number_of_bytes)
            with SMBus(1) as bus:
                bus.i2c_rdwr(read)
            break # break the trial loop if the above has not failed
        except: # if i2c communication fails
            if attempts >= 3:
                logger.warning("i2c communication failed 3 times while
writing to customer memory, skipping reading")
                return False # indicate that the writing process failed,
exit this function
            else:
                logger.error("i2c communication failed to read from
customer memory (" + str(attempts) + "/3)")
                attempts += 1
                print("Waiting 3 seconds...", end='\r')
                time.sleep(3) # avoid too close i2c communication, let
time to the sensor, may be busy

    reading = list(read)
    logger.debug("Reading from custom memory returned " + str(reading))
    return reading

def write_to_custom_memory(index, *bytes_to_write):
    """
    Write data to a custom memory address in the CO2 sensor internal memory
    :param index: index of the customer memory to write (see sensor doc)
    :param bytes_to_write: unlimited amount of bytes to write into the
internal custom memory at index (see sensor doc)
    :return: True (Success) or False (Fail)

```

```

"""
    logger.debug("Writing " + str(bytes_to_write) + " inside custom memory
at index " + str(hex(index)) + "...")
    crc8 = digest([index, *bytes_to_write]) # calculation of the CRC8
based on the index number and all the bytes sent
    attempts = 1 # trial counter for writing into the customer memory
    cycle = 1 # trial counter for i2c communication

    try:
        with SMBus(1) as bus:
            write = i2c_msg.write(CO2_address, [0x71, 0x54, index,
*bytes_to_write, crc8]) # see sensor doc
            bus.i2c_rdwr(write)
            logger.debug("i2c writing succeeded")
            # i2c writing function worked, and sensor didn't replied a NACK
on the SCK line
            # (see i2c working principle/theory)

        except:
            logger.critical("i2c failure while writing to custom memory")
            return False

        # check that the data are written correctly
        time.sleep(0.3)
        reading = read_from_custom_memory(index, len(bytes_to_write))
        if reading == [*bytes_to_write]: # because reading returns a list
            logger.debug("Success in writing " + str(bytes_to_write) + " inside
custom memory at index " + str(index))
            return reading # indicate that the writing process succeeded

        else:
            logger.error("Failed in writing " + str(bytes_to_write) + " inside
custom memory at index " + str(hex(index)))
            logger.debug("Value read is " + str(reading) + " in place of " +
str(bytes_to_write))

# -----
# Test Execution
# -----

# __name__ = '__main__' indicate that the Python sheet has been executed
directly
# in opposition with __name__ = '__CO2__' when the Python sheet is executed
as a library from another Python sheet

# What is below will be executed if user execute this Python code directly
($ python3 CO2.py)
# Code below is used to make trials to the CO2 sensor while developing

if __name__ == '__main__':
    now = datetime.now()
    logger.info("-----") # add a line in
the log file
    logger.info("Launching a new execution on the " +
str(now.strftime("%d/%m/%Y %H:%M:%S")))

    print("Reading internal timestamp")
    internal_timestamp()
    trigger_measurement(force=True)

    while True: # unstopped loop

```

```
get_data()  
print("waiting 10 seconds...")  
time.sleep(10) # wait 10 seconds
```

Annexe 6

OPCN3.py

```
#!/home/pi/seacanairy_project/venv/bin/python3
"""
Library for the use and operation of the Alphasense OPC-N3 sensor
"""
import codecs

import spidev # driver for the SPI/serial communication
import time
import struct # to convert the IEEE bytes to float
import datetime
import sys
import os # to create folders/files and read current path
from progress.bar import IncrementalBar # beautiful progress bar during
sampling
# import RPi.GPIO as GPIO # used for CS (Chip Select line)

import logging # save logger messages into memory

# yaml settings
import yaml # read user settings

# -----
# YAML SETTINGS
# -----

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close()

store_debug_messages = settings['CO2 sensor']['Store debug messages
(important increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

OPC_flushing_time = settings['OPC-N3 sensor']['Flushing time']

OPC_sampling_time = settings['OPC-N3 sensor']['Sampling time']

take_new_sample_if_checksum_is_wrong = \
    settings['OPC-N3 sensor'][
        'Take a new measurement if checksum is wrong (avoid shorter
sampling periods when errors)']
```



```
# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = keep a trace of some messages in a file to be reviewed
afterward (check for errors f-e)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s
%(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')

    logger = logging.getLogger('OPC-N3') # name of the logger
    # all further logging must be called by logger.'level' and not
logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/OPCN3-debugging.log" #
complete file location required for the Raspberry
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("DEBUG messages will be shown and stored in '" + str(log_file) +
""")
    logger = set_logger(message_level, log_file)
    # define a Handler which writes INFO messages or higher to the
sys.stderr/display
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)

else: # if this file is considered as a library (if you execute
'seacanairy.py' for example)
    # it will only print and store INFO messages and above in the
corresponding log_file
    if store_debug_messages:
        message_level = logging.DEBUG
    else:
        message_level = logging.INFO
    log_file = current_working_directory + "/" + project_name + "/" +
project_name + "-log.log"
    # no need to add a handler, because there is already one in
seacanairy.py
    logger = set_logger(message_level, log_file)
```

```

# -----
# SPI CONFIGURATION
# -----
# configuration of the Serial communication to the sensor

bus = 0 # name of the SPI bus on the Raspberry Pi 3B+, only one bus
device = 0 # name of the SS (Slave Selection) pin used for the OPC-N3
spi = spidev.SpiDev() # enable SPI (SPI must be enable in the RPi settings
beforehand)
spi.open(bus, device) # open the spi port at start
spi.max_speed_hz = 307200 # must be between 300 and 750 kHz
# Personal experiment shown that UART and SPI speeds must be multiple
# UART baud rate is 9600 for the GPS sensor
# 9600 * 2 * 2 * 2 * 2 * 2 * 2 = 307200
# If not, both sensor data are corrupted
# If not, OPCN3 returns alternately int(48) = hex(0x30) = bytes(00110000)
spi.mode = 0b01 # bytes(0b01) = int(1) --> SPI mode 1
# first bit (from right) = CPHA = 0 --> data are valid when clock is rising
# second bit (from right) = CPOL = 0 --> clock is kept low when idle
wait_10_milli = 0.015 # 15 ms
wait_10_micro = 1e-06
wait_reset_SPI_buffer = 3 # seconds
time_available_for_initiate_transmission = 10 # seconds - timeout for SPI
response

# if the sensor is disconnected, it can happen that the RPi wait for its
answer, which never comes...
# avoid the system to wait for unlimited time for that answer

# CS (chip selection) manually via GPIO - NOT CURRENTLY USED, to switch the
OPCN3 CS line manually up and down
# GPIO.setmode(GPIO.BCM) # use the GPIO names (GPIO1...) instead of the
processor pin name (BCM...)
# CS = 25 # GPIO number in which CS is connected
# GPIO.setup(CS, GPIO.OUT, initial=GPIO.HIGH)

# def cs_high(delay=0.010):
#     """Close communication with OPC-N3 by setting CS on HIGH"""
#     time.sleep(delay)
#     # GPIO.output(CS, GPIO.HIGH)
#     # time.sleep(delay)
#
#
# def cs_low(delay=0.010):
#     """Open communication with OPC-N3 by setting CS on LOW"""
#     time.sleep(delay)
#     # GPIO.output(CS, GPIO.LOW)
#     # time.sleep(delay)

def initiate_transmission(command_byte):
    """
    Initiate SPI transmission to the OPC-N3
    First loop on the manufacturer's flow Chart
    :param command_byte: byte to be sent during communication initiation
    :return: True when SPI initiation has been done, False if it failed
    """
    attempts = 0 # sensor is busy loop
    cycle = 1 # SPI buffer reset loop (going to the right on the

```

```
flowchart)
```

```

    logger.debug("Initiate transmission with command byte " +
str(hex(command_byte)))

    stop = time.time() + time_available_for_initiate_transmission
    # time in seconds at which we consider it took too much time to answer

    # cs_low() # not used anymore

    while time.time() < stop:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce
the amount of time between spi.xfer
        reading = spi.xfer([command_byte]) # initiate control of power
state

        # spi.xfer() means write a byte AND READ AT THE SAME TIME

        if reading == [243]: # SPI ready = 0xF3 = 243 = 0b11110011
            time.sleep(wait_10_micro)
            return True # indicate that the initiation succeeded

        if reading == [49]: # SPI busy = 0x31 = 49
            time.sleep(wait_10_milli)
            attempts += 1

        elif reading == [230] or reading == [99] or reading == [0]:
            # During developing, I noticed that these were the answers
given by the sensor when the CS line was
            # facing troubles.
            # This comes from personal experiment and not from the official
documentation
            # To resolve it, try connecting the CS line directly to the
ground (current setting)
            logger.critical("Problem with the SS (Slave Select) line "
                "(error code " + str(hex(reading[0])) + "),
skipping")
            cycle += 1
            logger.debug("Check that SS line is well kept DOWN (0V) during
transmission."
                " Try again by connecting SS Line of sensor to
Ground")
            print("Waiting SPI Buffer reset (" + str(reading) + ")",
end='\r')
            time.sleep(wait_reset_SPI_buffer)
            return False

        else:
            logger.critical(
                "Failed to initiate transmission (unexpected code returned:
" + str(hex(reading[0])) + ") (" + str(
                cycle) + "/3)")
            print("Waiting SPI Buffer reset (" + str(reading) + ")",
end='\r')
            time.sleep(wait_reset_SPI_buffer)
            cycle += 1 # increment of attempts
            attempts = 0

    if attempts > 60:
        # it is recommended to use > 20 in the Alphasense documentation
        # After experiment it seems that 60 is a good value
        # (does not take too much time, and let some chance to the

```

```

sensor to answer READY)
    logger.error("Failed 60 times to initiate control of power
state, reset OPC-N3 SPI buffer, trying again")
    # cs_high()
    print("Waiting SPI Buffer reset (" + str(reading) + ")",
end='\r')
    time.sleep(wait_reset_SPI_buffer) # time for spi buffer to
reset

    attempts = 0 # reset the "SPI busy" loop
    cycle += 1 # increment of the SPI reset loop
    # cs_low()

    if cycle >= 3:
        logger.critical("Failed to initiate transmission (reset 3 times
SPI, still error)")
        return False

    logger.critical("Transmission initiation timeout (> "
+ str(time_available_for_initiate_transmission) + "
secs)")
    return False # function depending on initiate_transmission function
will not continue, indicate error

def fan_off():
    """
    Turn OFF the fan of the OPC-N3
    :return: False if it succeeded turning off the fan, True if it failed
    """
    print("Turning fan OFF", end='\r')
    logger.debug("Turning fan OFF")
    attempts = 1

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce
the amount of time between spi.xfer
        if initiate_transmission(0x03):
            reading = spi.xfer([0x02])
            # cs_high()
            # spi.close() # close the serial port to let it available for
another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long
running time
            if reading == [0x03]: # official answer of the OPC-N3
                print("Fan is OFF")
                # time.sleep(0.5) # avoid too close communication (AND let
some time to the OPC-N3 to stop the fan)
                return False
            else:
                time.sleep(1) # let some time to the OPC-N3 (to try to
stop the fan)
                reading = read_DAC_power_status('fan')
                if reading == 0:
                    return False
                elif reading == 1:
                    attempts += 1
                    logger.warning("Failed to stop the fan, trying
again...")

                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)

```

```

        else:
            attempts += 1
            print("Waiting SPI Buffer reset", end='\r')
            time.sleep(wait_reset_SPI_buffer)
    if attempts >= 3:
        logger.critical("Failed 3 consecutive times to stop the
fan")
        return True
    else:
        logger.critical("Failed to stop the fan (transmission
problem)")
        return True
    return True

def fan_on():
    """
    Turn ON the fan of the OPC-N3
    :return: True if it succeeded turning off the fan, False if it failed
    """
    print("Turning fan ON", end='\r')
    logger.debug("Turning fan ON")

    attempts = 1

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce
the amount of time between spi.xfer
        if initiate_transmission(0x03):
            logger.debug("attempts = " + str(attempts))
            reading = spi.xfer([0x03])
            # cs_high()
            # spi.close() # close the serial port to let it available for
another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long
running time
            time.sleep(0.6) # wait > 600 ms to let the fan start
            if reading == [0x03]: # official answer of the OPC-N3
                print("Fan is ON")
                time.sleep(0.5) # avoid too close communication
                return True # indicate that fan has started
            else:
                time.sleep(1) # let time to the OPC-N3 to try to start the
fan

            reading = read_DAC_power_status('fan')
            if reading == 1:
                return True # indicate that fan has started
            elif reading == 0:
                logger.error("Failed to start the fan...")
                attempts += 1
                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)
            else:
                attempts += 1
                print("Waiting SPI Buffer reset", end='\r')
                time.sleep(wait_reset_SPI_buffer)
        if attempts >= 3:
            log = "Failed 3 consecutive times to start the fan"
            logger.critical(log)
            return False # indicate that fan is OFF
    else:

```

```

        logger.critical("Failed to start the fan (transmission
problem)")
        return False
    return True

def laser_on():
    """
    Turn ON the laser of the OPC-N3
    :return: True if it succeeded turning off the laser, False if it failed
    """
    print("Turning laser ON", end='\r')
    logger.debug("Turning laser ON")
    attempts = 0

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce
the amount of time between spi.xfer
        if initiate_transmission(0x03):
            reading = spi.xfer([0x07])
            # cs_high()
            # spi.close() # close the serial port to let it available for
another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long
running time
            if reading == [0x03]:
                print("Laser is ON          ")
                time.sleep(.5) # avoid too close communication
                return True # indicate that the laser is ON
            else:
                time.sleep(1) # let time to the OPC-N3 to try to start the
laser
                reading = read_DAC_power_status('laser')
                if reading == 1:
                    logger.info("Wrong answer received after SPI writing,
but laser is well on")
                    return True # indicate that the laser is ON
                elif reading == 0:
                    logger.error("Failed to start the laser, trying
again...")
                    attempts += 1
                    print("Waiting SPI Buffer reset", end='\r')
                    time.sleep(wait_reset_SPI_buffer)
                else:
                    attempts += 1
                    print("Waiting SPI Buffer reset", end='\r')
                    time.sleep(wait_reset_SPI_buffer)
            if attempts >= 3:
                logger.critical("Failed 3 consecutive times to start the
laser")
                return False # indicate that laser is still off
            else:
                logger.critical("Failed to start the laser (transmission
problem)")
                return False
    return False

def laser_off():
    """
    Turn OFF the laser of the OPC-N3

```

```

        :return: False if it succeeded turning off the laser, True if it failed
        """
    print("Turning the laser OFF", end='\r')
    logger.debug("Turning laser OFF")
    attempts = 0

    while attempts < 4:
        # logger.debug("attempts = " + str(attempts)) # disable to reduce
the amount of time between spi.xfer
        if initiate_transmission(0x03):
            reading = spi.xfer([0x06])
            # cs_high()
            # spi.close() # close the serial port to let it available for
another device
            # Avoid opening and closing ports too often.
            # Avoid getting "too much files opened" error after long
running time
            if reading == [0x03]:
                print("Laser is OFF")
                # time.sleep(1) # avoid too close communication
                return False
            else:
                time.sleep(1) # let time to the OPC-N3 to try to stop the
laser

                reading = read_DAC_power_status('laser')
                if reading == 0:
                    logger.info("Wrong answer received after writing, but
laser is well off")
                    return False
                elif reading == 1:
                    logger.error("Failed to stop the laser (code returned
is " + str(reading) + "), trying again...")
                    attempts += 1
                    print("Waiting SPI Buffer reset", end='\r')
                    time.sleep(wait_reset_SPI_buffer)
                else:
                    attempts += 1
                    print("Waiting SPI Buffer reset", end='\r')
                    time.sleep(wait_reset_SPI_buffer)
            if attempts >= 3:
                logger.critical("Failed 4 times to stop the laser")
                return True # indicate that laser is still on
        else:
            logger.critical("Failed to stop the laser (transmission
problem)")
            return True
    return True

def read_DAC_power_status(item='all'):
    """
    Read the status of the Digital to Analog Converter as well as the Power
Status
    Try only one time to read the byte(s)
    :param item: 'fan', 'laser', 'fanDAC', 'laserDAC', 'laser_switch',
'gain', 'auto_gain_toggle', 'all'
    :return: DAC power byte, 5 status bytes if argument is 'all'
    """
    print("Reading DAC power status", end='\r')
    if initiate_transmission(0x13):
        response = spi.xfer([0x13, 0x13, 0x13, 0x13, 0x13, 0x13])
        # cs_high()

```

```

    # spi.close() # close the serial port to let it available for
another device
    # Avoid opening and closing ports too often.
    # Avoid getting "too much files opened" error after long running
time
    time.sleep(0.5) # avoid too close communication

    if item == 'fan':
        logger.debug("DAC power status for " + str(item) + " is " +
str(response[0]))
        return response[0]
    elif item == 'laser':
        logger.debug("DAC power status for " + str(item) + " is " +
str(response[1]))
        return response[1]
    elif item == 'fanDAC':
        logger.debug("DAC power status for " + str(item) + " is " +
str(response[2]))
        response = 1 - (response[2] / 255) * 100 # see documentation
concerning fan pot
        logger.info("Fan is running at " + str(response) + "% (0 =
slow, 100 = fast)")
        return response
    elif item == 'laserDAC':
        logger.debug("DAC power status for " + str(item) + " is " +
str(response[3]))
        response = response[3] / 255 * 100 # see documentation
concerning laser pot
        logger.debug("Laser is at " + str(response) + "% of its maximal
power")
        return response
    elif item == 'laser_switch':
        logger.debug("DAC power status for " + str(item) + " is " +
str(response[4]))
        return response[4]
    elif item == 'gain':
        response = response[5] & 0x01
        logger.debug("DAC power status for " + str(item) + " is " +
str(response))
        return response
    elif item == 'auto_gain_toggle':
        response = response[5] & 0x02
        logger.debug("DAC power status for " + str(item) + " is " +
str(response))
        return response
    elif item is 'all':
        logger.debug("Full DAC power status is " + str(list(response)))
        return response
    else:
        raise ValueError("Argument of 'read_ADC_power_status' is
unknown, check your code!")

else:
    print("Waiting SPI Buffer reset", end='\r')
    time.sleep(wait_reset_SPI_buffer)
    return False # indicate an error

def digest(data):
    """
    Calculate the CRC8 Checksum with the bytes received
    :param data: list containing an infinite number of bytes with which to

```

```

calculate the checksum
:return: calculated checksum
"""
crc = 0xFFFF

for byteCtr in range(0, len(data)):
    to_xor = int(data[byteCtr])
    crc ^= to_xor
    for bit in range(0, 8):
        if (crc & 1) == 1:
            crc >>= 1
            crc ^= 0xA001
        else:
            crc >>= 1
# log = "Checksum is " + str(crc)
# logger.debug(log)
return crc & 0xFFFF

def check(checksum, *data):
    """
    Check that the data received are correct, based on those data and the
    checksum given
    :param checksum: checksum sent by the sensor (the last byte in any
    transmission)
    :param data: bytes sent by the sensor, with which to calculate the
    checksum
    :return: True if data are corrects, False if they are not
    """
    to_digest = []
    for i in data:
        to_digest.extend(i)
    if digest(to_digest) == join_bytes(checksum):
        log = "Checksum is correct"
        logger.debug(log)
        return True
    else:
        log = "Checksum is wrong"
        logger.debug(log)
        return False

def convert_IEEE754(value):
    """
    Join bytes and convert them to float according to the IEEE754
    encryption
    :param value: list containing the two bytes to decrypt
    :return: decrypted float
    """
    value = join_bytes(value)
    answer = struct.unpack('f', bytes(value))
    return answer

def loading_bar(name, delay):
    """
    Show a loading bar on the screen during a certain amount of time.
    Make the user understand the software is doing/waiting for something
    :param name: text to be shown on the left of the loading bar (waiting,
    sampling...)
    :param delay: amount of time the system is waiting (seconds)
    :return: nothing

```

```

"""
    bar = IncrementalBar(name, max=(2 * delay), suffix='% (elapsed)s/' +
str(delay) + ' seconds')
    for i in range(2 * delay):
        time.sleep(0.5)
        bar.next()
    bar.finish()
    return

def PM_reading():
    """
    Read the PM bytes only from the OPC-N3 sensor
    Read the data and convert them in readable format, checksum enabled
    Does neither start the fan nor start the laser
    Recommended to use read_histogram() instead of this function
    :return: List[PM 1, PM2.5, PM10]
    """
    print("YOU SHOULD BETTER USE OPCN3.read_histogram()")
    attempts = 1
    while attempts < 4:
        if initiate_transmission(0x32):
            PM_A = spi.xfer([0x32, 0x32, 0x32, 0x32])
            PM_B = spi.xfer([0x32, 0x32, 0x32, 0x32])
            PM_C = spi.xfer([0x32, 0x32, 0x32, 0x32])
            checksum = spi.xfer([0x32, 0x32])
            # spi.close()

            PM1 = round(struct.unpack('f', bytes(PM_A))[0], 3)
            PM25 = round(struct.unpack('f', bytes(PM_B))[0], 3)
            PM10 = round(struct.unpack('f', bytes(PM_C))[0], 3)

            if check(checksum, PM_A, PM_B, PM_C):
                print("PM 1:", PM1, "mg/m3\t|\tPM 2.5:", PM25,
"mg/m3\t|\tPM10:", PM10, "mg/m3")
                time.sleep(0.5) # avoid too close SPI communication
                return [PM1, PM25, PM10]
            if attempts >= 4:
                log = "PM data wrong 3 consecutive times, skipping PM
measurement"
                logger.critical(log)
                return ["error", "error", "error"]
            else:
                attempts += 1
                log = "Checksum for PM data is not correct, reading again
(" + str(attempts) + "/3)"
                logger.error(log)
                time.sleep(0.5) # avoid too close SPI communication

def getPM(flushing_time, sampling_time, start_fan_laser=True):
    """
    Get PM measurement from OPC-N3
    Recommended to use get_data() instead of this function
    :param flushing_time: time (seconds) during which the fan runs alone to
flush the sensor with fresh air
    :param sampling_time: time (seconds) during which the laser reads the
particulate matter in the air
    :return: List[PM1, PM2.5, PM10]
    """
    print("YOU SHOULD BETTER USE OPCN3.read_histogram()")
    try:

```

```
if start_fan_laser:
    fan_on()
    time.sleep(flushing_time)
    laser_on()
    print("Starting sampling") # will be printed on the same line as
"Laser is ON"
    time.sleep(sampling_time)
    PM = PM_reading()

    laser_off()
    fan_off()
except SystemExit or KeyboardInterrupt: # to stop the laser and the
fan in case of error or shutting down the program
    laser_off()
    fan_off()
    raise
return PM

def read_histogram(sampling_period, delete_previous=True):
    """
    Read all the available data from the OPC-N3
    It first read the histogram to remove the old data remaining in the
    OPCN3 buffer
    Then it let the sensor take sample during the defined sampling period
    Finally it read a last time the histogram data returned by the sensor
    It decode the bytes returned into readable format
    It returns everything in a dictionary
    :param: sampling_period: amount of time (seconds) during while the fan
    is running and laser is sampling
    :return: Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative
    humidity", "bin", "MToF", "sampling time",
    "sample flow rate", "reject count glitch", "reject count
    longTOF", "reject count ratio",
    "reject count out of range", "fan revolution count",
    "laser status"}
    """
    logger.debug("Reading histogram...")
    print("Reading histogram...", end='\r')

    # Create a dictionary containing data to be returned in case of error
    to_return = {
        "PM 1": "error",
        "PM 2.5": "error",
        "PM 10": "error",
        "temperature": "error",
        "relative humidity": "error",
        "sampling time": "error",
        "sample flow rate": "error",
        "reject count glitch": "error",
        "reject count long TOF": "error",
        "reject count ratio": "error",
        "reject count out of range": "error",
        "fan revolution count": "error",
        "laser status": "error",
        "bin 0": "error",
        "bin 1": "error",
        "bin 2": "error",
        "bin 3": "error",
        "bin 4": "error",
        "bin 5": "error",
        "bin 6": "error",
```

```

        "bin 7": "error",
        "bin 8": "error",
        "bin 9": "error",
        "bin 10": "error",
        "bin 11": "error",
        "bin 12": "error",
        "bin 13": "error",
        "bin 14": "error",
        "bin 15": "error",
        "bin 16": "error",
        "bin 17": "error",
        "bin 18": "error",
        "bin 19": "error",
        "bin 20": "error",
        "bin 21": "error",
        "bin 22": "error",
        "bin 23": "error",
        "bin 1 MToF": "error",
        "bin 3 MToF": "error",
        "bin 5 MToF": "error",
        "bin 7 MToF": "error"
    }

    # Delete old histogram data and start a new one
    if delete_previous:
        if initiate_transmission(0x30):
            answer = spi.xfer([0x00] * 86)
            logger.debug("SPI reading is:\r" + str(answer))
            # spi.close()
            logger.debug("Old histogram in the OPC-N3 deleted, starting a
new one")
        else:
            logger.critical("Failed to initiate histogram, skipping this
measurement")
            return to_return # indicate clearly an error in the data
recording

        delay = sampling_period * 2 # you must wait two times the
sampling_period in order that
        # the sampling time given by the OPC-N3 respects your sampling time
wishes
        # first 5 seconds are with low gain, and the next seconds are with high
gain (automatically performed by OPC-N3)
        print("                                ", end='\r') #
remove last line

        # Reading the histogram delete all the data in the OPCN3's buffer
        # If the checksum is wrong, seacanairy don't get the data as expected
        # Nevertheless, OPCN3 clean its buffer and all data are lost
        # So you must wait another x seconds to get sample
        if not take_new_sample_if_checksum_is_wrong:
            loading_bar('Sampling PM', delay)

        attempts = 1 # reset the counter for next measurement
        while attempts < 4:
            # If the user want to take a nex sample in case the checksum is
wrong (see explanation above), then
            # the system must wait the required amount of time in the reading
loop
            if take_new_sample_if_checksum_is_wrong:
                loading_bar('Sampling PM', delay)

```

```

if initiate_transmission(0x30):
    # read all the bytes and store them in a dedicated variable
    # see sensor documentation for more info
    bin = spi.xfer([0x00] * 48)
    MToF = spi.xfer([0x00] * 4)
    sampling_time = spi.xfer([0x00] * 2)
    sample_flow_rate = spi.xfer([0x00] * 2)
    temperature = spi.xfer([0x00] * 2)
    relative_humidity = spi.xfer([0x00] * 2)
    PM_A = spi.xfer([0x00] * 4)
    PM_B = spi.xfer([0x00] * 4)
    PM_C = spi.xfer([0x00] * 4)
    reject_count_glitch = spi.xfer([0x00] * 2)
    reject_count_longTOF = spi.xfer([0x00] * 2)
    reject_count_ratio = spi.xfer([0x00] * 2)
    reject_count_Out_Of_Range = spi.xfer([0x00] * 2)
    fan_rev_count = spi.xfer([0x00] * 2)
    laser_status = spi.xfer([0x00] * 2)
    checksum = spi.xfer([0x00] * 2)
    # spi.close()

    # check that the data transmitted are correct by comparing the
checksums
    # if the checksum is correct, then proceed...
    if check(checksum, bin, MToF, sampling_time, sample_flow_rate,
temperature, relative_humidity,
            PM_A, PM_B,
            PM_C, reject_count_glitch, reject_count_longTOF,
reject_count_ratio, reject_count_Out_Of_Range,
            fan_rev_count, laser_status):
        logger.debug("SPI reading is:\r" + str(bin) + " " +
str(MToF) + " " + str(sampling_time)
                    + " " + str(sample_flow_rate) + " " +
str(temperature) + " " + str(relative_humidity)
                    + " " + str(PM_A) + " " + str(PM_B) + " " +
str(PM_C) + " " + str(reject_count_glitch)
                    + " " + str(reject_count_longTOF) + " " +
str(reject_count_ratio) + " "
                    + str(reject_count_Out_Of_Range) + " " +
str(fan_rev_count)
                    + " " + str(laser_status))
        # return TRUE if the data are correct, and execute the
below

        # decode the bytes according to the IEEE 754 32 bytes
floating point format into decimals
        # rounding until 2 decimals, as this is the accuracy of the
OPC-N3 for PM values
        PM1 = round(struct.unpack('f', bytes(PM_A))[0], 2)
        PM25 = round(struct.unpack('f', bytes(PM_B))[0], 2)
        PM10 = round(struct.unpack('f', bytes(PM_C))[0], 2)
        print("PM 1:\t", PM1, " mg/m3", end="\t\t|\t")
        print("PM 2.5:\t", PM25, " mg/m3", end="\t\t|\t")
        print("PM 10:\t", PM10, " mg/m3")

        relative_humidity = round(100 *
(join_bytes(relative_humidity) / (2 ** 16 - 1)), 2)
        temperature = round(-45 + 175 * (join_bytes(temperature) /
(2 ** 16 - 1)), 2) # conversion in °C
        print("Temperature:", temperature, " °C (PCB Board)\t|
\tRelative Humidity:", relative_humidity,
            " %RH (PCB Board)")

```

```

        sampling_time = join_bytes(sampling_time) / 100
        print(" Sampling period:", sampling_time, "seconds",
end="\t\t|\t")
        sample_flow_rate = join_bytes(sample_flow_rate) / 100
        print(" Sampling flow rate:", sample_flow_rate, "mL/s |",
round(sample_flow_rate * 60, 2), "mL/min |",
        round(sample_flow_rate * 60 * 60 / 1000, 2), "L/h")
        # This is the amount of air passing through the laser beam,
not the total sampling flow rate!

        reject_count_glitch = join_bytes(reject_count_glitch)
        print(" Reject count glitch:", reject_count_glitch,
end="\t\t|\t")
        reject_count_longTOF = join_bytes(reject_count_longTOF)
        print(" Reject count long TOF:", reject_count_longTOF)
        reject_count_ratio = join_bytes(reject_count_ratio)
        print(" Reject count ratio:", reject_count_ratio,
end="\t\t|\t")
        reject_count_Out_Of_Range =
join_bytes(reject_count_Out_Of_Range)
        print(" Reject count Out Of Range:",
reject_count_Out_Of_Range)
        fan_rev_count = join_bytes(fan_rev_count)
        print(" Fan revolutions count:", fan_rev_count,
end="\t\t|\t")
        laser_status = join_bytes(laser_status)
        print(" Laser status:", laser_status)

    to_return = {
        "PM 1": PM1,
        "PM 2.5": PM25,
        "PM 10": PM10,
        "temperature": temperature,
        "relative humidity": relative_humidity,
        "sampling time": sampling_time,
        "sample flow rate": sample_flow_rate,
        "reject count glitch": reject_count_glitch,
        "reject count long TOF": reject_count_longTOF,
        "reject count ratio": reject_count_ratio,
        "reject count out of range": reject_count_Out_Of_Range,
        "fan revolution count": fan_rev_count,
        "laser status": laser_status,
        "bin 0": join_bytes(bin[0:1]),
        "bin 1": join_bytes(bin[2:3]),
        "bin 2": join_bytes(bin[4:5]),
        "bin 3": join_bytes(bin[6:7]),
        "bin 4": join_bytes(bin[8:9]),
        "bin 5": join_bytes(bin[10:11]),
        "bin 6": join_bytes(bin[12:13]),
        "bin 7": join_bytes(bin[14:15]),
        "bin 8": join_bytes(bin[16:17]),
        "bin 9": join_bytes(bin[18:19]),
        "bin 10": join_bytes(bin[20:21]),
        "bin 11": join_bytes(bin[22:23]),
        "bin 12": join_bytes(bin[24:25]),
        "bin 13": join_bytes(bin[26:27]),
        "bin 14": join_bytes(bin[28:29]),
        "bin 15": join_bytes(bin[30:31]),
        "bin 16": join_bytes(bin[32:33]),
        "bin 17": join_bytes(bin[34:35]),
        "bin 18": join_bytes(bin[36:37]),

```

```

        "bin 19": join_bytes(bin[38:39]),
        "bin 20": join_bytes(bin[40:41]),
        "bin 21": join_bytes(bin[42:43]),
        "bin 22": join_bytes(bin[44:45]),
        "bin 23": join_bytes(bin[46:47]),
        "bin 1 MToF": MToF[0]/3,
        "bin 3 MToF": MToF[1]/3,
        "bin 5 MToF": MToF[2]/3,
        "bin 7 MToF": MToF[3]/3,
    }

    print(" Bin number:\t", end='')
    for i in range(0, 24):
        print(to_return["bin " + str(i)], end=", ")
    print("") # go to next line
    print(" MToF:\t\t", end='')

    for i in range(0, 4):
        i = (i * 2) + 1
        print(to_return["bin " + str(i) + " MToF"], end=", ")
    print("") # go to next line

    if sampling_time > (sampling_period + 0.5): # we tolerate
a difference of 0.5 seconds
        log = "Sampling period of the sensor was " \
            + str(round(sampling_time - sampling_period, 2))
+ " seconds longer than expected"
        logger.warning(log)

    elif sampling_time < (sampling_period - 0.5):
        logger.warning("Sampling period of the sensor was "
            + str(round(sampling_period -
sampling_time, 2)) + " seconds shorter than expected")

    return to_return

else:
    # if the function with the checksum return an error (FALSE)
    logger.warning(
        "Error in the data received (wrong checksum), reading
    histogram again... (" + str(attempts) + "/3)")
    logger.warning("Data received were:\n" + str(bin) +
str(MToF) + str(sampling_time) +
                    str(sample_flow_rate) + str(temperature) +
                    str(relative_humidity) + str(PM_A) +
str(PM_B) +
                    str(PM_C) + str(reject_count_glitch) +
                    str(reject_count_longTOF) +
str(reject_count_ratio) +
                    str(reject_count_Out_Of_Range) +
str(fan_rev_count) +
                    str(laser_status) + str(checksum))
    print("Waiting SPI Buffer reset", end='\r')
    time.sleep(wait_reset_SPI_buffer) # let some times between
two SPI communications
        attempts += 1
    else:
        logger.critical("Failed to read histogram (transmission
    initiation problem)")
        return to_return

    if attempts >= 3:

```

```

        logger.error("Data were wrong 3 times (wrong checksum),
skipping this histogram reading")
        logger.warning("Data received were:\n" + str(bin) + str(MToF) +
str(sampling_time) +
                        str(sample_flow_rate) + str(temperature) +
                        str(relative_humidity) + str(PM_A) + str(PM_B) +
                        str(PM_C) + str(reject_count_glitch) +
                        str(reject_count_longTOF) +
str(reject_count_ratio) +
                        str(reject_count_Out_Of_Range) +
str(fan_rev_count) +
                        str(laser_status) + str(checksum))
        print("Waiting SPI Buffer reset", end='\r')
        time.sleep(wait_reset_SPI_buffer)
        return to_return

def get_data(flushing_time, sampling_time, start_fan_laser=True):
    """
    Get all the possible data from the OPC-N3 sensor
    Start the fan, flush air during defined time, start the laser,
    sample the air during defined time, turn off the laser and the fan
    :param flushing_time: time during which the ventilator is running
without sampling
                                to refresh the air inside the casing
    :param sampling_time: time during which the sensor is sampling
    :return: Dictionary{"PM 1", "PM 2.5", "PM 10", "temperature", "relative
humidity", "bin", "MToF", "sampling time",
                        "sample flow rate", "reject count glitch", "reject count
longTOF", "reject count ratio",
                        "reject count out of range", "fan revolution count",
"laser status"}
    """
    # return "error" everywhere in case of error during the measurement
(fan_on/laser_on/read_histogram...)
    # seacanairy.py need to find the items in the dictionary, if not if
crash
    to_return = {
        "PM 1": "error",
        "PM 2.5": "error",
        "PM 10": "error",
        "temperature": "error",
        "relative humidity": "error",
        "sampling time": "error",
        "sample flow rate": "error",
        "reject count glitch": "error",
        "reject count long TOF": "error",
        "reject count ratio": "error",
        "reject count out of range": "error",
        "fan revolution count": "error",
        "laser status": "error",
        "bin 0": "error",
        "bin 1": "error",
        "bin 2": "error",
        "bin 3": "error",
        "bin 4": "error",
        "bin 5": "error",
        "bin 6": "error",
        "bin 7": "error",
        "bin 8": "error",
        "bin 9": "error",
        "bin 10": "error",
    }

```

```

    "bin 11": "error",
    "bin 12": "error",
    "bin 13": "error",
    "bin 14": "error",
    "bin 15": "error",
    "bin 16": "error",
    "bin 17": "error",
    "bin 18": "error",
    "bin 19": "error",
    "bin 20": "error",
    "bin 21": "error",
    "bin 22": "error",
    "bin 23": "error",
    "bin 1 MToF": "error",
    "bin 3 MToF": "error",
    "bin 5 MToF": "error",
    "bin 7 MToF": "error"
}
try: # necessary to put an except condition (see below)
    if start_fan_laser:
        if not fan_on():
            logger.critical("Skipping histogram reading")
            return to_return
    print("Flushing fresh air", end='\r')
    time.sleep(flushing_time / 2)
    if start_fan_laser:
        if not laser_on():
            logger.critical("Skipping histogram reading")
            fan_off()
            return to_return
    print("Flushing fresh air", end='\r')
    time.sleep(flushing_time / 2)
    to_return = read_histogram(sampling_time)
    laser_off()
    fan_off()
    # spi.close()
    return to_return

except (KeyboardInterrupt, SystemExit): # in case of error AND if user
stop the software during sampling
    # Avoid that the laser and the fan keep running indefinitely if
system crash
    print(" ") # go to the next line
    logger.info("Python instance has been stopped, shutting laser and
fan OFF...")
    laser_off()
    fan_off()
    raise

def join_bytes(list_of_bytes):
    """
    Join bytes to an integer, from byte 0 to byte infinite (right to left)
    :param list_of_bytes: list [bytes coming from the spi.readbytes or
spi.xfer functions]
    :return: bytes concatenated to an integer
    """
    val = 0
    for i in reversed(list_of_bytes):
        val = val << 8 | i
    return val

```

```

def set_fan_speed(speed_percent):
    """
    Set the sensor fan speed
    Reduce fan speed can decrease dust deposition in the sensor casing
    Argument in percent, calibrated from the slowest as possible to the
    fastest

    :param speed_percent: number between 0 and 100 (0 = slowest, 100 =
    fastest)
    :return: nothing
    """
    if speed_percent < 0 or speed_percent > 100:
        raise ValueError("Fan speed of OPC-N3 sensor must be a number
        between 0 and 100 (0 = slowest, 100 = fastest)")
    value = int((45 + speed_percent / 100 * 55) / 100 * 255)
    # Personal investigations shows that the fan don't work below 45%
    # Formula makes a calculation to convert 0% as 45% --> easier for user
    input
    if initiate_transmission(0x42):
        reading = spi.xfer([0, value])
        logger.info("Fan speed is set on " + str(speed_percent) + " (0 =
        the slowest, 100 = the fastest)")
    else:
        logger.error("Failed to set the fan speed")

def initialization_SPI():
    """
    Initialize the OPCN3 SPI system
    To be executed once after Seacanairy power up
    To be executed on time only after powering up the OPCN3
    :return: nothing
    """
    print("Initializing OPCN3 SPI...", end='\r')

    # Make any communication to start the Sensor SPI
    # Personal investigations shows that first communication is always lost
    answer = []
    if initiate_transmission(0x3F):
        answer += spi.xfer([0x3F])
        for _ in range(63):
            answer += spi.xfer([0x3F])
            if answer[-2:] == [0x42, 0x53]:
                pass
        string = ''
        for x in range(len(answer)):
            string += chr(answer[x])

        print("OPCN3 infostring: '" + str(string) + "'")

    return

if __name__ == '__main__':
    # The code below runs if you execute this code from this file (you must
    execute OPC-N3 and not seacanairy)
    while True:
        logger.debug("Code is running from the OPC-N3 file itself, debug
        messages shown")
        # fan_on()
        # read_DAC_power_status('fan')

```

```
# time.sleep(1)
# laser_on()
# read_DAC_power_status('laser')
# time.sleep(1)
# laser_off()
# read_DAC_power_status('laser')
# time.sleep(1)
# fan_off()
# read_DAC_power_status('fan')
# print("sleep")
# time.sleep(3)

get_data(2, 3)
print("sleep")
time.sleep(5)
```

Annexe 7

AFE.py

```
from datetime import datetime
import time
import os.path
import yaml
import logging
import sys
import threading
from progress.bar import IncrementalBar # to show beautiful loading bar on
the screen during sampling

# -----
# I2C
# -----
from smbus2 import SMBus
from sys import exit

# emplacement variable
bus = SMBus(1)

# attributed canals and associated emplacements variable
address = 0b1110110

# -----
# YAML SETTINGS
# -----

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close()

store_debug_messages = settings['AFE Board']['Store debug messages
(important increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward
(check for errors fe)
```

```
def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s
%(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')

    logger = logging.getLogger('AFE Board') # name of the logger
    # all further logging must be called by logger.'level' and not
logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/Alphasense_board-
debugging.log"
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("Alphasense Board DEBUG messages will be shown and stored in '" +
str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # define a Handler which writes INFO messages or higher to the
sys.stderr/display
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)

else: # if this file is considered as a library (if you execute
'seacanairy.py' for example)
    # it will only print and store INFO messages and above in the
corresponding log_file
    if store_debug_messages:
        message_level = logging.DEBUG
    else:
        message_level = logging.INFO
    log_file = '/home/pi/seacanairy_project/log/' + project_name + '-
log.log' # complete location needed on the RPI
    # no need to add a handler, because there is already one in
seacanairy.py
    logger = set_logger(message_level, log_file)

# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as 'ROOT' and NOT 'GPS'

# -----
# ADC channels
# -----
```

```
# Channel Address - Single channel use
# See LTC2497 data sheet, Table 3, Channel Selection.
# All channels are uncommented - comment out the channels you do not plan
to use.

channel0 = 0xB0
channel1 = 0xB8
channel2 = 0xB1
channel3 = 0xB9
channel4 = 0xB2
channel5 = 0xBA
channel6 = 0xB3
channel7 = 0xBB
channel8 = 0xB4
channel9 = 0xBC
channel10 = 0xB5
channel11 = 0xBD
channel12 = 0xB6
channel13 = 0xBE
channel14 = 0xB7
channel15 = 0xBF

# reference voltage of the ADC
vref = 5

# To calculate the voltage, the number read in is 3 bytes. The first bit is
ignored.
# Max reading is 2^23 or 8,388,608
max_reading = 8388608.0

# lange = number of bytes to read. A minimum of 3 bytes are read in.
# In this sample we read in 6 bytes, ignoring the last three bytes
# zeit = tells how frequently you want the readings to be read from the
ADC.
# Define the time to sleep between the readings.
# tiempo = shows how frequently each channel is read in over the I2C bus.
# Best to use tiempo between each successive readings.

lange = 0x06 # number of bytes to read in the block
zeit = 15 # number of seconds to sleep between each measurement
sleep = 0.2 # number of seconds to sleep between each channel reading

# has to be more than 0.2 (seconds)

# -----
# CALIBRATION INFORMATION
# -----

# Temperature
with open(current_working_directory +
'/AFE_calibration/temperature_calib.yaml') as file:
    temp_calib = yaml.safe_load(file)
    file.close()

# NO2
with open(current_working_directory + '/AFE_calibration/NO2_calib.yaml') as
file:
    NO2_calib = yaml.safe_load(file)
    file.close()

# SO2
with open(current_working_directory + '/AFE_calibration/SO2_calib.yaml') as
```

HZS

```
file:
    SO2_calib = yaml.safe_load(file)
    file.close()

# OX
with open(current_working_directory + '/AFE_calibration/OX_calib.yaml') as
file:
    OX_calib = yaml.safe_load(file)
    file.close()

# CO
with open(current_working_directory + '/AFE_calibration/CO_calib.yaml') as
file:
    CO_calib = yaml.safe_load(file)
    file.close()

def getADCreading(adc_address, adc_channel):
    """
    Read tension from the ADC on a certain channel

    :param adc_address: slave i2c address
    :param adc_channel: channel where to read tension
    :return: tension between channel and ground (volts)
    """

    attempts = 0

    while attempts < 4:

        try:
            bus.write_byte(adc_address, adc_channel)
            # print("Reading tension...
", end='\r')
            time.sleep(sleep)
            reading = bus.read_i2c_block_data(adc_address, adc_channel,
lange)

            # ----- Start conversion for the Channel Data -----
            valor = (((reading[0] & 0x3F) << 16)) + ((reading[1] << 8)) +
(((reading[2] & 0xE0)))
            # add a debug function
            # debug(print("Valor is 0x%x" % valor))

            # ----- End of conversion of the Channel -----
            volts = round(valor * vref / max_reading, 7)
            # Rounding to 7 decimals because ADC accuracy is 3.9 microvolt
            # print("Reading tension...", volts, "V", end='\r')

            if (reading[0] & 0b11000000) == 0b11000000:
                logger.error(
                    "Input voltage is either open or more than " +
str(vref) + "Volts.")
                logger.warning("The reading may not be correct. Value read
is " + str(volts) + " mV")

                # time.sleep(sleep) # be sure to have some time laps between
two I2C reading/writing # i2c don't care!
                return volts

        except:
            if attempts >= 3:
                logger.critical("i2c transmission failed 3 consecutive
```

```

times(" + str(sys.exc_info())
      + "), skipping i2c reading")
    return False # indicate clearly that system has failed

    logger.error("Error in the i2c transmission (" +
str(sys.exc_info())
      + "), trying again... (" + str(attempts) + "/3)")
    attempts += 1 # increment of reading_trials
    time.sleep(1) # if transmission fails, wait a bit to try again
(sensor is maybe busy)

    return False

#
=====

reading_multiplier = 1000 # multiplication of the value given by the rpi

def read_temp():
    """
    Measure tension of the temperature sensor
    (Note that sensor is not located in the gas hood.)

    :return: Dictionary containing tension in milli volts {'temperature
raw'}
    """

    volts = getADCreading(address, channel5)
    if volts is not False:
        tempv = round(reading_multiplier * volts, 5)
        logger.debug("Tension from temperature sensor (AFE board) is " +
str(tempv) + " mV")
        time.sleep(sleep)

        temp_to_return = {
            "temperature raw": tempv,
            "temperature": "-"
        }
    else:
        logger.critical("Failed to read temperature")
        temp_to_return = {
            "temperature raw": "error",
            "temperature": "error"
        }

    return temp_to_return

def read_NO2():
    """
    Measure tension of NO2 main and auxiliary electrodes

    :return: Dictionary containing tensions in milli volts {'NO2 main',
'NO2 aux'}
    """
    volts = getADCreading(address, channel8)
    if volts is not False:
        NO2v_main = round(reading_multiplier * volts, 5)

```



```
logger.debug("Tension from NO2 sensor (main) is " + str(NO2v_main)
+ " mV")
time.sleep(sleep)
NO2v_aux = round(reading_multiplier * getADCreading(address,
channel4), 5)
logger.debug("Tension from NO2 sensor (aux) is " + str(NO2v_aux) +
" mV")
time.sleep(sleep)

NO2_to_return = {
    "NO2 main": NO2v_main,
    "NO2 aux": NO2v_aux,
    "NO2 ppb": "-"
}
else:
logger.critical("Failed to read NO2 sensor")
NO2_to_return = {
    "NO2 main": "error",
    "NO2 aux": "error",
    "NO2 ppb": "error"
}

return NO2_to_return

def read_OX():
    """
    Measure tension of OX main and auxiliary electrodes

    :return: Dictionary containing tensions in milli volts {'OX main', 'OX
aux'}
    """
    volts = getADCreading(address, channel7)
    if volts is not False:
        Oxv_main = round(reading_multiplier * volts, 5)
        logger.debug("Tension from Ox sensor (main) is " + str(Oxv_main) +
" mV")
        time.sleep(sleep)
        Oxv_aux = round(reading_multiplier * getADCreading(address,
channel13), 5)
        logger.debug("Tension from Ox sensor (aux) is " + str(Oxv_aux) + "
mV")
        time.sleep(sleep)

        OX_to_return = {
            "OX main": Oxv_main,
            "OX aux": Oxv_aux,
            "OX ppb": "-"
        }
    else:
        logger.critical("Failed to read OX")
        OX_to_return = {
            "OX main": "error",
            "OX aux": "error",
            "OX ppb": "error"
        }

    return OX_to_return

def read_SO2():
```

```

"""
    Measure tension of SO2 main and auxiliary electrodes

    :return: Dictionary containing tensions in milli volts {'SO2 main',
'SO2 aux'}
"""
    volts = getADCreading(address, channel6)
    if volts is not False:
        SO2v_main = round(reading_multiplier * volts, 5)
        logger.debug("Tension from SO2 sensor (main) is " + str(SO2v_main)
+ " mV")
        time.sleep(sleep)
        SO2v_aux = round(reading_multiplier * getADCreading(address,
channel2), 5)
        logger.debug("Tension from SO2 sensor (aux) is " + str(SO2v_aux) +
" mV")
        time.sleep(sleep)

        SO2_to_return = {
            "SO2 main": SO2v_main,
            "SO2 aux": SO2v_aux,
            "SO2 ppb": "-"
        }

    else:
        logger.critical("Failed to read SO2")

        SO2_to_return = {
            "SO2 main": "error",
            "SO2 aux": "error",
            "SO2 ppb": "error"
        }

    return SO2_to_return

def read_CO():
    """
    Measure tension of CO main and auxiliary electrodes

    :return: Dictionary containing tensions in milli volts {'CO main', 'CO
aux'}
"""
    volts = getADCreading(address, channel10)
    if volts is not False:
        COv_main = round(reading_multiplier * volts, 5)
        time.sleep(sleep)
        logger.debug("Tension from CO sensor (main) is " + str(COv_main) +
" mV")
        COv_aux = round(reading_multiplier * getADCreading(address,
channell1), 5)
        logger.debug("Tension from CO sensor (aux) is " + str(COv_aux) + "
mV")
        time.sleep(sleep)

        CO2_to_return = {
            "CO main": COv_main,
            "CO aux": COv_aux,
            "CO ppb": "-"
        }

    else:
        logger.critical("Failed to read CO")

```

```
        CO2_to_return = {
            "CO main": "error",
            "CO aux": "error",
            "CO ppb": "error"
        }

    return CO2_to_return

def calibrate_temperature(main):
    """
    Apply calibration to the temperature measurement
    :param main: electric tension from the temperature sensor
    :return: temperature in °C
    """
    temperature = ((main - temp_calib["Vkal"]) / temp_calib["Thermal
sensitivity"]) \
        + temp_calib["Tkal"]

    temperature = round(temperature, 2)

    return temperature

def calibrate_NO2(main, aux):
    """
    Apply calibration to the NO2 measurements
    :param main: electrical tension from the main electrode
    :param aux: electrical tension from the auxiliary electrode
    :return: NO2 concentration (ppb)
    """
    # Algorithm 1
    NO2_ppb = (
        (
            (main - NO2_calib["WE0_e"]) -
            (
                NO2_calib["nt"] * (aux -
NO2_calib["AE0_e"])
            )
        )
        / NO2_calib["WE_SENS"]
    ) + NO2_calib["C"]

    NO2_ppb = round(NO2_ppb, 1)

    return NO2_ppb

def calibrate_OX(main, aux, NO2_ppb):
    """
    Apply calibration to the OX measurements
    :param main: electrical tension from the main electrode
    :param aux: electrical tension from the auxiliary electrode
    :param NO2_ppb: NO2 concentration from the other sensor
    :return: OX concentration (ppb)
    """
    # Algorithm 3
    O3_ppb = (
        (
            (main - OX_calib["WE0_e"] - (NO2_ppb *
```

```

OX_calib["WE_SENS_NO2"])
        ) -
        (OX_calib["WE0_s"] - OX_calib["AE0_s"]) -
        (
            OX_calib["nt"] *
            (aux - OX_calib["AE0_e"])
        )
    ) / OX_calib["WE_SENS"]
) \
+ OX_calib["C"]

O3_ppb = round(O3_ppb, 1)

return O3_ppb

def calibrate_SO2(main, aux):
    """
    Apply calibration to the SO2 measurements
    :param main: electrical tension from the main electrode
    :param aux: electrical tension from the auxiliary electrode
    :return: SO2 concentration (ppb)
    """
    # Algorithm 4
    SO2_ppb = -1 * \
        (
            (
                (main - SO2_calib["WE0_e"]) -
                SO2_calib["WE0_s"] - SO2_calib["nt"]
            )
            / SO2_calib["WE_SENS"]) \
        + SO2_calib["C"]

    SO2_ppb = round(SO2_ppb, 1)

    return SO2_ppb

def calibrate_CO(main, aux):
    """
    Apply calibration to the CO measurements
    :param main: electrical tension from the main electrode
    :param aux: electrical tension from the auxiliary electrode
    :return: CO concentration (ppb)
    """
    # Algorithm 1
    CO_ppb = (
        (
            (main - CO_calib["WE0_e"]) -
            (
                CO_calib["nt"] *
                (aux - CO_calib["AE0_e"])
            )
        )
        / CO_calib["WE_SENS"]) \
        + CO_calib["C"]

    CO_ppb = round(CO_ppb, 1)

    return CO_ppb

```

HZS

```
def calibrate_all(data):
    """
    Apply calibration to all available items
    :param data: dictionary containing values to calibrate
    :return: dictionary with calibrated items
    """

    if "temperature raw" in data:
        data["temperature"] = calibrate_temperature(data["temperature
raw"])

    if "NO2 main" in data:
        data["NO2 ppb"] = calibrate_NO2(data["NO2 main"], data["NO2 aux"])
        if "OX main" in data:
            data["OX ppb"] = calibrate_OX(data["OX main"], data["OX aux"],
data["NO2 ppb"])

    if "SO2 main" in data:
        data["SO2 ppb"] = calibrate_SO2(data["SO2 main"], data["SO2 aux"])

    if "CO main" in data:
        data["CO ppb"] = calibrate_CO(data["CO main"], data["CO aux"])

    return data

def get_data():
    """
    Get all available data from the 4-AFE Alphasense Board (one signe
instantaneous reading)

    :return: dictionary{'NO2 ppb', 'NO2 main', 'NO2 aux', 'OX ppb', 'OX
main', 'OX aux',
                        'SO2 ppb', 'SO2 main', 'SO2 aux', 'CO ppb', 'CO main', 'CO
aux',
                        'temperature', 'temperature raw'}
    """

    data = {}

    data.update(read_NO2())
    data.update(read_OX())
    data.update(read_SO2())
    data.update(read_CO())
    data.update(read_temp())
    data = calibrate_all(data)

    return data

def print_measurements(data):
    """
    Print all measurement data on the screen
    :param data: dictionary containing all the data
    :return: nothing
    """

    print("\t\t ppb \t\t( main (mV), aux (mV) )")
    print("NO2:\t\t", data["NO2 ppb"], "\t\t(", data["NO2 main"], ", ",
data["NO2 aux"], ")")
    print("OX:\t\t", data["OX ppb"], "\t\t(", data["OX main"], ", ",
data["OX aux"], ")")
    print("SO2:\t\t", data["SO2 ppb"], "\t\t(", data["SO2 main"], ", ",
```

```

data["SO2 aux"], ")")
    print("CO:\t\t", data["CO ppb"], "\t\t(", data["CO main"], ", ",
data["CO aux"], ")")
    print("Temperature:\t", data["temperature"], "\t\t(", data["temperature
raw"], ")")

    return

def start_averaged_data(number_of_measurements, delay = 0, display=True):
    """
    Perform multiple readings and makes an average
    Run get_averaged_data() once thread is finished to get the data
    Improved for threading application (no display prints)

    :param number_of_measurements: number of measurement to average, each
single measurement taking around 2 seconds
    :return: Dictionary{'NO2 main', 'NO2 aux', 'OX main', 'OX aux',
        'SO2 main', 'SO2 aux', 'CO main', 'CO aux',
        'temperature raw'}
    """
    global thread_data

    thread_data = {}
    if delay:
        time.sleep(delay)

    if display:
        bar = IncrementalBar("Reading tensions", max=(5 *
number_of_measurements))

    logger.debug("Starting averaged data reading")
    NO2_main = []
    NO2_aux = []

    OX_main = []
    OX_aux = []

    SO2_main = []
    SO2_aux = []

    CO_main = []
    CO_aux = []

    temperature_main = []

    for _ in range(number_of_measurements):
        NO2 = read_NO2()
        if display: bar.next()
        NO2_main += [NO2['NO2 main']]
        NO2_aux += [NO2['NO2 aux']]

    thread_data.update({"NO2 main min": min(NO2_main)})
    thread_data.update({"NO2 main max": max(NO2_main)})
    thread_data.update({"NO2 aux min": min(NO2_aux)})
    thread_data.update({"NO2 aux max": max(NO2_aux)})

    sum = 0
    for i in range(len(NO2_main)):
        sum += NO2_main[i]
    NO2_main = sum / len(NO2_main)

```

```
sum = 0
for i in range(len(NO2_aux)):
    sum += NO2_aux[i]
NO2_aux = sum / len(NO2_aux)

for _ in range(number_of_measurements):
    OX = read_OX()
    if display: bar.next()
    OX_main += [OX['OX main']]
    OX_aux += [OX['OX aux']]

thread_data.update({"OX main min": min(OX_main)})
thread_data.update({"OX main max": max(OX_main)})

thread_data.update({"OX aux min": min(OX_aux)})
thread_data.update({"OX aux max": max(OX_aux)})

sum = 0
for i in range(len(OX_main)):
    sum += OX_main[i]
OX_main = sum / len(OX_main)

sum = 0
for i in range(len(OX_aux)):
    sum += OX_aux[i]
OX_aux = sum / len(OX_aux)

for _ in range(number_of_measurements):
    SO2 = read_SO2()
    if display: bar.next()
    SO2_main += [SO2['SO2 main']]
    SO2_aux += [SO2['SO2 aux']]

thread_data.update({"SO2 main min": min(SO2_main)})
thread_data.update({"SO2 main max": max(SO2_main)})
thread_data.update({"SO2 aux min": min(SO2_aux)})
thread_data.update({"SO2 aux max": max(SO2_aux)})

sum = 0
for i in range(len(SO2_main)):
    sum += SO2_main[i]
SO2_main = sum / len(SO2_main)

sum = 0
for i in range(len(SO2_aux)):
    sum += SO2_aux[i]
SO2_aux = sum / len(SO2_aux)

for _ in range(number_of_measurements):
    CO = read_CO()
    if display: bar.next()
    CO_main += [CO['CO main']]
    CO_aux += [CO['CO aux']]

thread_data.update({"CO main min": min(CO_main)})
thread_data.update({"CO main max": max(CO_main)})
thread_data.update({"CO aux min": min(CO_aux)})
thread_data.update({"CO aux max": max(CO_aux)})

sum = 0
for i in range(len(CO_main)):
    sum += CO_main[i]
```

```

CO_main = sum / len(CO_main)

sum = 0
for i in range(len(CO_aux)):
    sum += CO_aux[i]
CO_aux = sum / len(CO_aux)

for _ in range(number_of_measurements):
    temp = read_temp()
    if display: bar.next()
    temperature_main += [temp['temperature raw']]

thread_data.update({"temperature min": min(temperature_main)})
thread_data.update({"temperature max": max(temperature_main)})

sum = 0

for i in range(len(temperature_main)):
    sum += temperature_main[i]
temperature = sum / len(temperature_main)

thread_data.update({
    'NO2 main': NO2_main,
    'NO2 aux': NO2_aux,
    'OX main': OX_main,
    'OX aux': OX_aux,
    'SO2 main': SO2_main,
    'SO2 aux': SO2_aux,
    'CO main': CO_main,
    'CO aux': CO_aux,
    'temperature raw': temperature
})

thread_data = calibrate_all(thread_data)

logger.debug("Execute function 'get_averaged_data' to show data on
screen")

bar.finish()

return thread_data

def start_background_average_measurement(number_of_measurements, delay=0):
    """
    Start a new thread to perform averaged reading in the background
    Run get_averaged_data() once thread is finished to get the data

    thread = threading.Thread(target=AFE.start_averaged_data,
args=([number_of_measurements, delay]), daemon=True)
in your own code is preferred

:param number_of_measurements: number of measurement to average, each
single measurement taking around 2 seconds
:param delay: amount of time in between the start of the thread and the
start of the sampling operation
:return: nothing
    """

    print("Starting background AFE average reading...")
    x = threading.Thread(target=start_averaged_data,
args=([number_of_measurements, delay]), daemon=True)

```



```
x.start()
logger.debug("Execute function 'get_averaged_data' to show data on
screen")
return

def get_averaged_data():
    """
    Read the data of the last start_averaged_data() (or
    start_background_average_measurement()) performed

    :return: dictionary{'NO2 ppb', 'NO2 main', 'NO2 aux', 'OX ppb', 'OX
    main', 'OX aux',
                        'SO2 ppb', 'SO2 main', 'SO2 aux', 'CO ppb', 'CO main', 'CO
    aux',
                        'temperature', 'temperature raw'}
    """
    global thread_data

    print_measurements(thread_data)

    return thread_data

# open the file where the data will be stored
if __name__ == "__main__":
    # Execute an execution test if the script is executed from there
    while True:
        get_data()
```

Annexe 8

GPS.py

```
"""
Library for the use of the U-BLOX-7 GNSS module (Velleman VMA430)
Get the data from the UART port
Convert the NMEA protocol and extract the useful information
Should work with other UART GNSS devices
"""

from datetime import datetime, timezone

import serial # UART libraries, to install this library: pip3 install
pyserial
import time
import yaml
import logging
# import RPi.GPIO as GPIO
import sys
import os.path

# -----
# YAML SETTINGS
# -----

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close()

store_debug_messages = settings['GPS']['Store debug messages (important
increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward
(check for errors fe)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='% (asctime)s % (name)-12s % (levelname)-8s
% (message) s',
```

```

        datefmt='%d-%m %H:%M',
        filename=log_file,
        filemode='a')

    logger = logging.getLogger('GPS') # name of the logger
    # all further logging must be called by logger.'level' and not
logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/GPS-debugging.log"
    if not os.path.isfile(log_file):
        os.mkknod(log_file)
    print("GPS DEBUG messages will be shown and stored in '" +
str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # define a Handler which writes INFO messages or higher to the
sys.stderr/display
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)

else: # if this file is considered as a library (if you execute
'seacanairy.py' for example)
    # it will only print and store INFO messages and above in the
corresponding log_file
    if store_debug_messages:
        message_level = logging.DEBUG
    else:
        message_level = logging.INFO
    log_file = '/home/pi/seacanairy_project/log/' + project_name + '-
log.log' # complete location needed on the RPI
    # no need to add a handler, because there is already one in
seacanairy.py
    logger = set_logger(message_level, log_file)

# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as 'ROOT' and NOT 'GPS'

# -----
# GPIO SETTINGS
# -----
# Used during developing to synchronize the UART reading with the GPS pulse
# spi.readall() make this sep unnecessary
# GPIO.setmode(GPIO.BCM)
# GPIO.setup(25, GPIO.OUT, initial=GPIO.LOW, pull_up_down=GPIO.PUD_DOWN)
# GPIO.output(25, GPIO.LOW)

```

```

# def pulse():
# GPIO.output(25, GPIO.HIGH)
# time.sleep(.2)
# GPIO.output(25, GPIO.LOW)

def get_raw_reading(close_UART=True):
    """
    Get raw GPS reading via UART
    Read all the lines available on the UART port
    :return: raw data from the GPS
    """
    global ser
    port = '/dev/ttyAMA0'
    try:
        # USB = '/dev/ttyACM0'
        # PL011 = '/dev/serial0' == '/dev/ttyAMA0'
        logger.debug("Port used for UART communication is: " + str(port))
        ser = serial.Serial(port=port, baudrate=9600)
        print("Starting UART communication...", end='\r')
        time.sleep(1)
        ser.flush()
    try:
        print("Synchronizing... ", end='\r')
        ser.read_all() # delete all corrupted data
        ser.flush() # flush the buffer
        time.sleep(1)
        reading = ser.read_all()
        if close_UART:
            ser.close() # avoid unnecessary port closing if second
reading is requested
    except:
        logger.critical("Failed to read GPS data on UART port " +
str(port) + " (" + str(sys.exc_info()) + ")")
        ser.close()
        return False # indicate error
    except:
        logger.critical("Failed to initiate UART port " + str(port) + " ("
+ str(sys.exc_info()) + ")")
        return False # indicate error

    reading = str(reading, 'utf-8', errors='replace') # convert the text
sent in b'...' format into readable format...
    # it will also skip the line where the GPS propose it (see NMEA
protocol)
    # 'replace' = replace the unencodable unicode to a question mark
    logger.debug("Raw reading is:\r" + str(reading[:-1]))
    return reading

def lat_long_decode(raw_position, compas):
    """
    Decode longitude and latitude data from NMEA into readable format
    :param raw_position: raw longitude/latitude word
    :param compas: compas (N/S/W/E)
    :return: string(decoded latitude/longitude)
    """
    position = raw_position.split(".")
    min = position[0][-2:]
    min_dec = position[1]
    deg = position[0][0:-2]

```

```

position = deg + '°' + min + "." + min_dec + "' " + compas
return position

def decode_NMEA(data):
    """
    Decode the NMEA script and get the useful data
    Only the necessary data are extracted from the frames
    :param data: whole string returned by the GPS (all the lines of the
    NMEA)
    :return: Dictionary{fix time, fix date, fix date and time, latitude,
    longitude, SOG, COG, status,
    horizontal precision, altitude, WGS84 correction, current
    time, accuracy}
    """
    data = data.split("\r\n") # create a list of lines (\r\n is sent by
    the sensor at the end of each line)
    to_return = {}
    visible_satellites = 0
    for i in range(len(data)): # don't know at which line data will be
    send, so it will search for the good line
        print("Decode data...", end='\r')
        # print(data[i], end='\r')
        # time.sleep(.1) # let a bit of time for the user to see the data
    returned by the GPS
        # print("
    ", end='\r')
        if data[i][0:6] == "$GPRMC":
            if check(data[i]):
                GPRMC = data[i].split(",")
                fix_time = GPRMC[1][0:2] + ":" + GPRMC[1][2:4] + ":" +
    GPRMC[1][4:6]
                date = GPRMC[9][0:2] + "-" + GPRMC[9][2:4] + "-" +
    GPRMC[9][4:6]
                to_return.update({
                    "current date and time": date + " " + fix_time + "
    UTC",
                    "current date": date,
                    "current time": fix_time,
                })
                if GPRMC[2] == "V": # indicate that GPS is not working
    good
                    logger.warning("GPS does not receive signal")
                    to_return.update({
                        "status": "NOK",
                        "latitude": "no fix",
                        "longitude": "no fix",
                        "SOG": "no fix",
                        "COG": "no fix",
                        "altitude": "no fix",
                        "WGS84 correction": "no fix",
                        "fix status": "no fix",
                        "horizontal precision": "no fix",
                        "accuracy": "no fix"
                    })
                    return to_return
                elif GPRMC[2] == "A": # indicate that GPS is working fine
                    latitude = lat_long_decode(GPRMC[3], GPRMC[4])
                    longitude = lat_long_decode(GPRMC[5], GPRMC[6])
                    SOG = GPRMC[7].replace(',', ' .')
                    COG = GPRMC[8]

```

```

        to_return.update({
            "latitude": latitude,
            "longitude": longitude,
            "SOG": SOG,
            "COG": COG,
            "status": "OK"
        })

    else:
        logger.critical("Something wrong with the GPRMC data,
GPS satus returned is: " + str(GPRMC[2]))

    elif data[i][0:6] == "$GPGGA":
        if check(data[i]):
            GPGGA = data[i].split(",")
            current_time = GPGGA[1][0:2] + ":" + GPGGA[1][2:4] + ":" +
GPGGA[1][4:6] + " UTC"
            altitude = GPGGA[9] + " m"
            WGS84_correction = GPGGA[11] + " " + GPGGA[12]
            position_fix_status_indicator = GPGGA[6]
            horizontal_precision = float(GPGGA[8])
            accuracy = ''
            if horizontal_precision < 2:
                accuracy = "very good"
            elif 2 <= horizontal_precision < 3:
                accuracy = "good"
            elif 3 <= horizontal_precision < 5:
                accuracy = "average"
            elif 5 <= horizontal_precision < 6:
                accuracy = "poor"
            elif horizontal_precision >= 6:
                accuracy = "very poor"
            if position_fix_status_indicator == '0':
                fix_status = "No fix/invalid"
            elif position_fix_status_indicator == '1':
                fix_status = "Standard GPS 2D/3D"
            elif position_fix_status_indicator == '2':
                fix_status = "DGPS"
            elif position_fix_status_indicator == '6':
                fix_status = "DR"
            else:
                logger.error(
                    "Unknown position fix status indicator in GPGGA: "
+ str(position_fix_status_indicator))
                fix_status = "Unknown: " +
str(position_fix_status_indicator)

            to_return.update({
                "altitude": altitude,
                "WGS84 correction": WGS84_correction,
                "fix status": fix_status,
                "current time": current_time,
                "horizontal precision": horizontal_precision,
                "accuracy": accuracy
            })

    elif data[i][0:6] == "$GPGSV":
        visible_satellites += 1

    to_return.update({
        "available satellites": visible_satellites
    })

```

```
    return to_return

def digest(string_line):
    """
    Calculate the checksum based on the transmitted data
    Put the whole NMEA line in the argument, function will automatically
    remove the checksum at the end
    COPY-PASTED AND ADAPTED FROM WIKIPEDIA
    :param NMEAstring: line of data transmitted by the GPS
    :return:
    """
    calc_cksum = 0
    NMEAstring = string_line[1:-3]
    for s in NMEAstring:
        # it is XOR of each Unicode integer representation
        calc_cksum ^= ord(s)

    calc_cksum = str(hex(calc_cksum))[2:] # get hex representation
    calc_cksum = calc_cksum.upper() # convert the lowercase to uppercase
    (abc to ABC)
    # if not, Python does not recognize this string as a hexadecimal
    return calc_cksum

def check(NMEA_line):
    """
    Check that the data transmitted are correct
    Put the whole line in the argument, function extract the checksum at
    the end on its own
    :param NMEA_line: one line of data transmitted by the GPS
    :return: True (data are corrects), False (data are not corrects)
    """
    calc = digest(NMEA_line)
    checksum = NMEA_line[-2:] # extract the checksum, the two last
    characters
    if calc == checksum:
        logger.debug("Checksum is correct")
        return True
    else:
        logger.warning("Checksum is not correct: calculation is " +
str(calc) + " | sensor's checksum is " + str(checksum))
        logger.warning("NMEA line was: " + str(NMEA_line))
        return False

def get_position():
    """
    Read position data from the GPS receiver

    :return: Dictionary{fix time, fix date, fix date and time, latitude,
    longitude, SOG, COG, status,
    horizontal precision, altitude, WGS84 correction, current
    time, accuracy, fix status}
    """
    logger.debug("Get position")

    global ser
    attempts = 1

    to_return = {
```

```

    "current date and time": "",
    "current date": "",
    "current time": "",
    "latitude": "error",
    "longitude": "error",
    "SOG": "error",
    "COG": "error",
    "status": "error",
    "horizontal precision": "error",
    "altitude": "error",
    "WGS84 correction": "error",
    "fix status": "error",
    "accuracy": "error",
    "available satellites": 0
} # you must return all those items to avoid bugs in seacanairy.py (f-
e looking for an item which doesn't exist)

while attempts <= 4:
    reading = get_raw_reading(close_UART=False)
    if not reading: # if it failed to read UART, it returns False
        logger.critical("Unable to read GPS sensor, skipping reading")
        ser.close() # if no more reading necessary, close UART port
        return to_return # return a dictionary full of "error"
    else:
        try: # avoid errors because of 'I don't know why the sensor
sometimes delete items in the NMEA at random'
            data = decode_NMEA(reading) # decode the raw reading
            to_return.update(data) # update the dictionary with the
data the function got
            logger.debug("'to_return' is:\r" + str(to_return))
            # At each trial, it will update the dictionary
        except:
            logger.error("There were an error while decoding NMEA
protocol (" + str(str(sys.exc_info())) + ")")

            if "error" in to_return.values(): # if the dictionary contains
an error, try again
                attempts += 1
                if attempts >= 4: # if the system has tried 3 times to
read the data but that there are still errors
                    logger.error("Tried 3 times to get full GPS data, still
a value 'error'")
                    ser.close() # if no more reading necessary, close UART
port
                    break # exit the loop and print the data anyway
                    logger.warning("Data missing in GPS transmission, reading
again (" + str(attempts) + "/3)")
                    time.sleep(.2)
                else: # if there are no errors, then exit the loop and proceed
                    ser.close() # if no more reading necessary, close UART
port
                    break

    print("Current date and time:\t", to_return["current date and time"])
    print("Latitude:\t", to_return["latitude"], "\t|\tLongitude:\t",
to_return["longitude"])
    print("Altitude:\t", to_return["altitude"], "\t|\t|\tWGS84 correction:",
to_return["WGS84 correction"])
    print("SOG:\t|\t", to_return["SOG"], "kts", "\t|\t|\tCOG:\t|\t ", end='')
    if to_return["COG"] == '':
        print("no speed")
    else:

```



```
        print(to_return["COG"])
        print("Horizontal deviation:\t", to_return["horizontal precision"])
        print("GPS mode:\t", to_return["fix status"])
        print("Accuracy:\t", to_return["accuracy"], "\t\t|\tGPS status:\t",
to_return["status"])
        print("Available satellites:\t", to_return["available satellites"])

    return to_return

if __name__ == '__main__':
    print("GPS.py is running alone")
    while True:
        get_position()
        time.sleep(5)
```

Annexe 9

flow.py

```
# get the time
import time
from datetime import date, datetime

# Get the errors
import sys

# Create folders and files
import os

# smbus2 is the new smbus, allow more than 32 bits writing/reading
from smbus2 import SMBus, i2c_msg
# 'SMBus' is the general driver for i2c communication
# 'i2c_msg' allow to make i2c write followed by i2c read WITHOUT any STOP
byte (see sensor documentation)

# logging
import logging

# yaml settings
import yaml

# progress bar during sampling
from progress.bar import IncrementalBar

# take measurement while doing something else
import threading

# I2C address of the CO2 device
air_address = 1
O2_address = 2
CO2_address = 3
N2O_address = 4
Ar_address = 5

# emplacement variable
bus = SMBus(1) # make it easier to read/write to the sensor (bus.read or
bus.write...)

# -----
# YAML SETTINGS
# -----
# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
```

HZS

```
settings = yaml.safe_load(file)
file.close() # close the file after use

store_debug_messages = settings['Air flow sensor']['Store debug messages
(important increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward
# (check for errors fe)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s
%(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')

    logger = logging.getLogger('Flow meter') # name of the logger
    # all further logging must be called by logger.'level' and not
    logging.'level'
    # if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
    return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/flow_meter-debugging.log"
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("Flow meter DEBUG messages will be shown and stored in '" +
str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # The following HANDLER must be activated ONLY if you run this code
alone
    # Without the 'if __name__ == '__main__' condition, all the logging
messages are displayed 3 TIMES
    # (once for the handler in CO2.py, once for the handler in OPCN3.py,
and once for the handler in seacanairy.py)

    # define a Handler which writes INFO messages or higher to the
sys.stderr/display (= the console)
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)
```

```

else: # if this file is considered as a library (if you execute
seacanairy.py for example)
    # if the user asked to store all the messages in
'seacanairy_settings.yaml'
    if store_debug_messages:
        message_level = logging.DEBUG
    # if the user don't want to store everything
    else:
        message_level = logging.INFO
    # Create a file to store the log if it doesn't exist yet
    log_file = current_working_directory + "/" + project_name + "/" +
project_name + "-log.log"
    logger = set_logger(message_level, log_file)
    # no need to add a handler, because there is already one in
seacanairy.py

# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as ROOT and NOT 'CO2 sensor'

# -----

def loading_bar(name, delay):
    """
    Show a loading bar on the screen during a a certain amount of time
    Make the user understand the software is doing/waiting for something
    :param name: Text to be shown on the left of the loading bar
    :param length: Amount of time the system is waiting in seconds
    :return: nothing
    """
    bar = IncrementalBar(name, max=(2 * delay), suffix='% (elapsed)s/' +
str(delay) + ' seconds')
    for i in range(2 * delay):
        time.sleep(0.5)
        bar.next()
    bar.finish()
    return

def digest(buf):
    """
    Calculate the CRC8 checksum (based on the CO2 documentation example)
    :param buf: List[bytes to digest]
    :return: checksum
    """
    # Translation of the C++ code given in the documentation
    crcVal = 0x00
    _from = 0 # the first item in a list is named 0
    _to = len(buf) # if there are two items in the list, then len() return
1 --> range(0, 1) = 2 loops

    for i in range(_from, _to):
        curVal = buf[i]

        for j in range(0, 8): # C++ stops when J is not < 8 --> same for
python in range
            if ((crcVal ^ curVal) & 0x80) != 0:
                crcVal = (crcVal << 1) ^ 0x31

            else:

```

```

        crcVal = (crcVal << 1)

        curVal = (curVal << 1) # this line is in the "for j" loop, not
in the "for i" loop

        checksum = crcVal & 0xff # keep only the 8 last bits

        return checksum

def check(checksum, data):
    """
    Check that the data transmitted are correct using the data and the
    given checksum
    :param checksum: Checksum given by the sensor (see sensor doc)
    :param data: List[bytes to be used in the checksum calculation (see
    sensor doc)]
    :return: True if the data are correct, False if not
    """
    calculation = digest(data)
    if calculation == checksum:
        logger.debug("CRC8 is correct, data are valid")
        return True
    else:
        logger.debug("CRC8 does not fit, data are wrong")
        logger.error("Checksum is wrong, sensor checksum is: " +
str(checksum) +
                    ", seacanairy checksum is: " + str(calculation) +
                    ", data returned by the sensor is:" + str(data))
        if data[0] and data[1] == 0:
            logger.debug("Sensor returned 0 values, it is not ready,
waiting a bit")
            print("Sensor not ready, waiting...", end='\r')
            time.sleep(3)
        return False

def check(checksum, data):
    """
    Check that the data transmitted are correct using the data and the
    given checksum
    :param checksum: Checksum given by the sensor (see sensor doc)
    :param data: List[bytes to be used in the checksum calculation (see
    sensor doc)]
    :return: True if the data are correct, False if not
    """
    calculation = digest(data)
    if calculation == checksum:
        logger.debug("CRC8 is correct, data are valid")
        return True
    else:
        logger.debug("CRC8 does not fit, data are wrong")
        logger.error("Checksum is wrong, sensor checksum is: " +
str(checksum) +
                    ", seacanairy checksum is: " + str(calculation) +
                    ", data returned by the sensor is:" + str(data))
        if data[0] and data[1] == 0:
            logger.debug("Sensor returned 0 values, it is not ready,
waiting a bit")
            print("Sensor not ready, waiting...", end='\r')
            time.sleep(3)
        return False

```

```

def get_data(print_data=True):
    """
    Get flow measurement from the Sensirion mass flow meter 4100
    :return: dictionary {"flow [sccm]", "flow [slm]", "flow [slh]}
    """
    logger.debug("Reading flow from Sensirion Mass Flow Meter Sensor")
    if print_data:
        print("Reading flow...", end='\r')

    to_return = {
        "flow [sccm]": "error",
        "flow [slm]": "error",
        "flow [slh]": "error"
    }

    attempts = 1

    while attempts <= 4:
        if attempts >= 3:
            logger.critical("i2c transmission failed 3 consecutive times,
skipping this flow reading")
            return to_return
        try:
            answer = bus.read_i2c_block_data(air_address, 0xF1, 3)
            logger.debug("i2c succeeded, answer is: " + str(answer))
            if check(answer[2], answer[0:2]):
                break
        except:
            attempts += 1
            logger.error("i2c communication failed while reading flow (" +
str(sys.exc_info()) + ")")

    if answer[0] == 255:
        flow_sccm = 0
        flow_slm = 0
        flow_slh = 0
    else:
        flow_sccm = (answer[0] << 8) + answer[1]
        flow_slm = flow_sccm / 1000
        flow_slh = round(flow_slm * 60, 2)

    if print_data:
        print("                                ", end='\r')
        print(flow_sccm, "\tsccm [~= mL/min]")
        print(flow_slm, "\tslm [~= L/min]")
        print(flow_slh, "\tslh [~= L/h]")

    to_return.update({
        "flow [sccm]": flow_sccm,
        "flow [slm]": flow_slm,
        "flow [slh]": flow_slh
    })

    return to_return

def start_averaged_measurement(sampling_period,
number_of_measurement_during_sampling_period, delay=0):
    global sccm
    global slm

```

```
global slh
sccm = []
slm = []
slh = []

time.sleep(delay)
sleep = sampling_period / number_of_measurement_during_sampling_period
for _ in range(number_of_measurement_during_sampling_period):
    reading = get_data(print_data=False)
    sccm.append(reading['flow [sccm]'])
    slm.append(reading['flow [slm]'])
    slh.append(reading['flow [slh]'])
    time.sleep(sleep)

def get_averaged_measurement():

    to_return = {
        "average flow [sccm]": "error",
        "average flow [slm]": "error",
        "average flow [slh]": "error",
    }

    global sccm
    global slm
    to_return.update({"slm min": min(slm), "slm max": max(slm)})
    global slh
    try:
        sum = 0
        for i in range(len(sccm)):
            sum += sccm[i]
        sccm = round(sum/len(sccm), 0)

        sum = 0
        for i in range(len(slm)):
            sum += slm[i]
        slm = round(sum/len(slm), 3)

        sum = 0
        for i in range(len(slh)):
            sum += slh[i]
        slh = round(sum/len(slh), 2)

    except:
        logger.error("Error occurred while computing average flow rate (" +
str(sys.exc_info()) + ")")
        return to_return

    print("Average flow rate:")
    print(sccm, "\tsccm [~= mL/min]")
    print(slm, "\tslm [~= L/min] (min:", to_return["slm min"], "max:",
to_return["slm max"], ")")
    print(slh, "\tslh [~= L/h]")

    to_return.update({
        "average flow [sccm]": sccm,
        "average flow [slm]": slm,
        "average flow [slh]": slh,
    })

    return to_return
```

```
if __name__ == "__main__":  
    while True:  
        get_data()  
        time.sleep(1)
```


Annexe 10

database.py

```
import yaml
import logging
import sys
import mysql.connector
import os

# -----
# YAML SETTINGS
# -----

# Get current directory
current_working_directory = str(os.getcwd())

with open(current_working_directory + '/seacanairy_settings.yaml') as file:
    settings = yaml.safe_load(file)
    file.close()

store_debug_messages = settings['MySQL database settings']['Store debug
messages (important increase of logs)']

project_name = settings['Seacanairy settings']['Sampling session name']

host = settings['MySQL database settings']['Host']
user = settings['MySQL database settings']['User']
password = settings['MySQL database settings']['Password']
db_name = settings['MySQL database settings']['Database name']
table_name = project_name

# -----
# LOGGING SETTINGS
# -----
# all the settings and other code for the logging
# logging = tak a trace of some messages in a file to be reviewed afterward
# (check for errors fe)

def set_logger(message_level, log_file):
    # set up logging to file
    logging.basicConfig(level=message_level,
                        format='%(asctime)s %(name)-12s %(levelname)-8s
%(message)s',
                        datefmt='%d-%m %H:%M',
                        filename=log_file,
                        filemode='a')
```

```

logger = logging.getLogger('MySQL') # name of the logger
# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as 'ROOT' and NOT 'OPC-N3'
return logger

if __name__ == '__main__': # if you run this code directly ($ python3
CO2.py)
    message_level = logging.DEBUG # show ALL the logging messages
    # Create a file to store the log if it doesn't exist
    log_file = current_working_directory + "/log/mysql-debugging.log"
    if not os.path.isfile(log_file):
        os.mknod(log_file)
    print("MySQL DEBUG messages will be shown and stored in '" +
str(log_file) + "'")
    logger = set_logger(message_level, log_file)
    # define a Handler which writes INFO messages or higher to the
sys.stderr/display
    console = logging.StreamHandler()
    console.setLevel(message_level)
    # set a format which is simpler for console use
    formatter = logging.Formatter('%(name)-12s: %(levelname)-8s
%(message)s')
    # tell the handler to use this format
    console.setFormatter(formatter)
    # add the handler to the root logger
    logging.getLogger().addHandler(console)

else: # if this file is considered as a library (if you execute
'seacanairy.py' for example)
    # it will only print and store INFO messages and above in the
corresponding log_file
    if store_debug_messages:
        message_level = logging.DEBUG
    else:
        message_level = logging.INFO
    log_file = '/home/pi/seacanairy_project/log/' + project_name + '-
log.log' # complete location needed on the RPI
    # no need to add a handler, because there is already one in
seacanairy.py
    logger = set_logger(message_level, log_file)

# all further logging must be called by logger.'level' and not
logging.'level'
# if not, the logging will be displayed as 'ROOT' and NOT 'MySQL'

# -----
# CODE
# -----

# Global variables
global connected
connected = False
table_created = False
global data_in_cache
data_in_cache = False
global db_line_count
db_line_count = False
global datafile_length
datafile_length = False

```

```
# Connect to the server
```

```
print("Connecting to MySQL database...", end='\r')

def connect(print_status=False):
    """
    Check connection and connect if connection is lost
    :param print_status: decide to show connection status or not
    :return: True (connected) or False (Not connected)
    """
    global mydb # share variable through this code file
    global connected

    print("Checking MySQL connection status...", end='\r')

    if connected: # if connection has already been established before
        return connected
    else: # if first execution, or if connection has failed last time
        try:
            mydb = mysql.connector.connect(
                host=host,
                user=user,
                password=password,
                database=db_name)
            connected = mydb.is_connected()
            if connected:
                if print_status:
                    logger.info("Connected to database for online data
storage")
                return connected
            else:
                if print_status:
                    logger.warning(
                        "No connection to database. Check internet status
and database information in the settings")
                return connected
        except:
            if print_status:
                logger.error("Failed to connect to database (" +
str(sys.exc_info()) + ")")
            connected = False
            return connected

def create_new_table(header_list, header_type):
    """
    Create a new table in the database
    :param header_list: list of all the column headers
    :return:
    """
    global connected
    global table_created
    header_name = str(header_list[0]) + " " + header_type[0]
    for i in range(1, len(header_list)):
        header_name += ", " + header_list[i] + " " + header_type[i]

    if connect():
        try:
            print("Creating database table '" + str(table_name) + "'...",
end='\r')
```

```

        mycursor = mydb.cursor()
        mycursor.execute("CREATE TABLE IF NOT EXISTS `" +
str(table_name) + "` (" + header_name + ")")
        logger.info("Table (already) created (" + str(table_name) +
")")

        table_created = True
        return True
    except:
        logger.error("Failed to create new table in the database (" +
str(sys.exc_info()) + ")")
        connected = False
        return False

def number_of_lines_in_db():
    """
    Count the number of lines in the current table
    :return: number of lines
    """
    global connected
    try:
        mycursor = mydb.cursor()
        mycursor.execute("SELECT COUNT(*) FROM `" + str(table_name) + "`")
        result = mycursor.fetchone()
        print("There is already", result[0], "lines in db")
        return result[0]
    except:
        connected = False
        logger.error("Failed to count the number of lines in the database
(" + str(sys.exc_info()) + ")")
        return 0

def upload_data(header_list, list_of_lines):
    """
    Append data to the table in the database
    :param header_list: List of the headers in which to write some data
    :param data: List containing the data to fill in those headers, in good
order
    :return: True or False (success or failure)
    """
    global db_line_count
    # Convert header list in string
    # Necessary to create the MySQL function
    header_name = str(header_list[0])
    for i in range(1, len(header_list)):
        header_name += ", " + str(header_list[i])

    header_name = header_name.replace('"', '') # no guillemet for header
creation (see MySQL theory)

    # print("Header name is:", header_name)
    # print("Len header is:", len(header_list))
    # print("len of lines to send is:", len(list_of_lines))

    for i in range(len(list_of_lines)):
        # Create data string from list
        to_write = []
        for j in list_of_lines[i]:
            if isinstance(j, list):
                to_write += j
            else:

```

```

        to_write += [j]

    to_upload = '' + to_write[0] + '' # the first one will always be
the datetime, requires guillemets

    for j in range(1, len(to_write)):
        if to_write[j] == '-' or to_write[j] == 'error' or to_write[j]
== '-\n' or to_write[j] == 'error\n' or \
            to_write[j] == 'no fix':
            to_upload += ',NULL'
        else:
            try:
                to_upload += ',' + str(float(to_write[j].replace('\n',
'')))
            except:
                to_upload += ', ' + str(to_write[j].replace('\n', ''))
+ ''

    # print("To upload is:", to_upload)
    # print("Len to_upload is:", len(data))
    sql = "-"
    print("Sending", str(i) + '/' + str(len(list_of_lines)), " lines to
MySQL database...", end='\r')

    if "0x00" in to_upload: # if line in file is corrupted, skip this
line and go ahead
        try:
            sql = "INSERT INTO `" + str(table_name) + "`"
            mycursor = mydb.cursor()
            mycursor.execute(sql)
            mydb.commit()
            db_line_count += 1 # one line sent, database is now one
line bigger
        except:
            pass
        continue
    try:
        # print("INSERT INTO `" + str(table_name) + "` (" + header_name
+ ") VALUES (" + str(to_upload) + ")")
        mycursor = mydb.cursor()
        sql = "INSERT INTO `" + str(table_name) + "` (" + header_name +
") VALUES (" + str(
            to_upload) + ")"
        mycursor.execute(sql)
        mydb.commit()
        db_line_count += 1 # one line sent, database is now one line
bigger
    except:
        logger.error("Failed to save data into the database (" +
str(sys.exc_info()) + "; line was: ", str(sql),
        ")")
        sql = "INSERT INTO `" + str(table_name) + "`"
        mycursor = mydb.cursor()
        mycursor.execute(sql)
        mydb.commit()
        db_line_count += 1 # one line sent, database is now one line
bigger
    print("
", end='\r')
    print(len(list_of_lines), "line(s) uploaded on the database")

```

```

def csv_file_length():
    """
    Get data file length
    :return: data file length
    """
    global datafile_length
    file = open(current_working_directory + "/" + str(project_name) + "/" +
str(project_name) + "-data.csv", 'r')
    datafile_length = 0
    for line in file:
        if line != "\n":
            datafile_length += 1
    file.close()
    del file # remove from memory
    return datafile_length - 1 # minus 1 for the column header line

def update(header_list, header_type, data_to_add):
    """
    Upload data on the database if possible
    :return:
    """
    global table_created
    global db_line_count
    global datafile_length
    global connected

    if not datafile_length:
        datafile_length = csv_file_length()
        # print("file length is:", datafile_length)

    if connect(): # check connection, and go ahead if ok
        if not table_created:
            try:
                table_created = create_new_table(header_list, header_type)
            except:
                print("Impossible to create table (" + str(sys.exc_info())
+ ")")
                connected = False
                return # impossible to store data if table doesn't exist
        if not db_line_count:
            db_line_count = number_of_lines_in_db()
            # print("db length is:", db_line_count)

        for i in range(1, len(data_to_add)):
            if type(data_to_add[i]) == str:
                # in case of any sensor error, it return either "-" or
"error"
                # a float value could become a string, leading to database
failure
                data_to_add[i].replace('-', 'NULL').replace('error',
'NULL').replace('no fix', 'NULL')
                # in MySQL, NULL means 'empty'

            if db_line_count == datafile_length: # if the same amount of data
in the db and in the csv file
                upload_data(header_list, [data_to_add]) # send only the newest
data

        else:
            logger.info("Sending pending data to MySQL database")
            to_upload = []

```

```
        csv_file = open(
            current_working_directory + "/" + str(project_name) + "/" +
str(project_name) + "-data.csv", 'r')
        csv_lines = csv_file.readlines() # load all the lines in the
memory

        csv_file.close()
        for x in range(db_line_count + 1, datafile_length + 1):
            to_upload += [csv_lines[x].split(',')]
            # for i in range(1, len(csv_lines)):
            #     if type(data_to_add[i]) == str:
            #         csv_lines[i].replace('-',
'NULL').replace('error', 'NULL')
            to_upload += [data_to_add]
            upload_data(header_list, to_upload)
            del csv_lines # remove this huge variable from RAM
            del csv_file

    else:
        return

    datafile_length += 1
    # if this function is executed, it means that new data have been taken
    # therefore, we know the data file will have one line more
    # this way, we avoid loading each time the csv file
    # same increment after every database upload
```

Annexe 11

seacanairy_settings.yaml

```
# SEACANAIRY CONFIGURATION FILE

# Recommendations
# do not change the file syntax
# settings must either be Yes or No
# numbers must be integer (without decimals)
# After changing any settings, check that the Software still work

Seacanairy settings:
# Name of the folder and database table in which the log and data files
will be stored:
# WARNING:
Sampling session name: "essai-17-08-bis"
# Amount of time between each consecutive measurement
Sampling period: 60 # seconds
Activate M&C air pump: Yes
Air pump minimum running time: 5 # seconds per loop

MySQL database settings:
Activate database upload: Yes
Store debug messages (important increase of logs): No
Host: "remotemysql.com"
User: "4tgGwNUHei"
Password: "iz3EKsfjBU"
Database name: "4tgGwNUHei"

CO2 sensor:
Activate this sensor: Yes
Automatic sampling frequency (number of sample during the above sampling
period): 1
Amount of time required for the sensor to take the measurement: 5 #
seconds (default value: 10 seconds)
Store debug messages (important increase of logs): No
Number of reading attempts: 6 # default value: 6

OPC-N3 sensor:
Activate this sensor: Yes
# Amount of time at which the fan keep running to refresh the air inside
the sensor casing
Flushing time: 0
# Amount of time at which the laser is kept on and measure the air
# This period will be multiplied by 2 in practice because the sensor
automatically take a
```


HZS

```
# first measurement in high gain and then another one in low gain mode
Sampling time: 4
Fan speed: 100 # 0 = the slowest, 100 = the fastest
# In case of data transmission error, take another sample (Yes) or
# read the data again even if sampling time is really short (No)
Take a new measurement if checksum is wrong (avoid shorter sampling
periods when errors): Yes
Store debug messages (important increase of logs): No
```

Air flow sensor:

```
Activate this sensor: Yes
Store debug messages (important increase of logs): No
```

GPS:

```
Activate this sensor: Yes
Store debug messages (important increase of logs): No
```

AFE Board:

```
Activate this sensor: Yes
Store debug messages (important increase of logs): No
# Perform multiple readings and average them to reduce noise
Absorption time between air pump stop and reading: 2
Noise reduction - number of reading averaged: 4
# Reading occurs after minimum running time
```

Annexe 12

AFE calibration

Files are similar for other gas sensors (SO₂; NO₂, CO, OX, and temperature).

```
# ALPHASENSE 4-AFE CALIBRATION FILE
# CO

Calibration information:
  Name: "Lukas"
  Version number: "1.0"

# Algorithm no.1
WE_SENS: 0.2776
WE0_e: 305.5
AE0_e: 301
WE0_s: -28.24
AE0_s: 25.76
WE0: 277.26
AE0: 326.76

nt: -0.9
C: 0 # maybe an error there, confusion between excel file and paper
formula
```

Annexe 13

set_system_time.sh

```
#!/bin/sh

printf "Check the system time: "
date
printf "Is the current date and time correct? [Y/n] " ; read -r answer
if [ $answer == "Y" ] ;
then
    printf "Exiting time setting"
    sleep 1
    exit
fi
if [ $answer == "n" ] ;
then
    printf "RTC time is: "
    sudo hwclock -r
    printf "Is that time correct? [Y/n] " ; read -r answer
fi
if [ $answer == "Y" ] ;
then
    printf "Applying RTC time to the system"
    sudo hwclock -s
    printf "\nSystem time is now "
    date
    printf "Exiting this shell script\n"
    sleep 1
    exit
fi
if [ $answer == "n" ] ;
then
    printf "Is the computer connected to the internet? [Y/n] " ; read -r
answer
    if [ $answer == "Y" ] ;
    then
        printf "Let some time to the system to get time from the internet"
        printf "\nShell script will close\nExecute this script again in one
minute\n"
        sleep 5
        exit
    fi
    if [ $answer == "n" ] ;
    then
        printf "Type hereafter the date in the following format: YYYY-MM-DD
(2001-09-11): " ; read -r date_input
        sudo date +%F -s "$date_input"
```

```
    printf "Type now the current time in the following format: hh:mm:ss
(12:30:55): " ; read -r time_input
    sudo date +%T -s "$time_input"
    printf "Date and Time are now: "
    date
    printf "\nWriting current time inside RTC..."
    sudo hwclock -w
    printf "Exiting this shell script\n"
    sleep 1
    exit
fi
fi
exit
```

Annexe 14

Graph from the measuring device

Several samplings has been achieved using the Seacanairy in its final stage. Following graphs comes from one of those sampling sessions.

1 Temperature

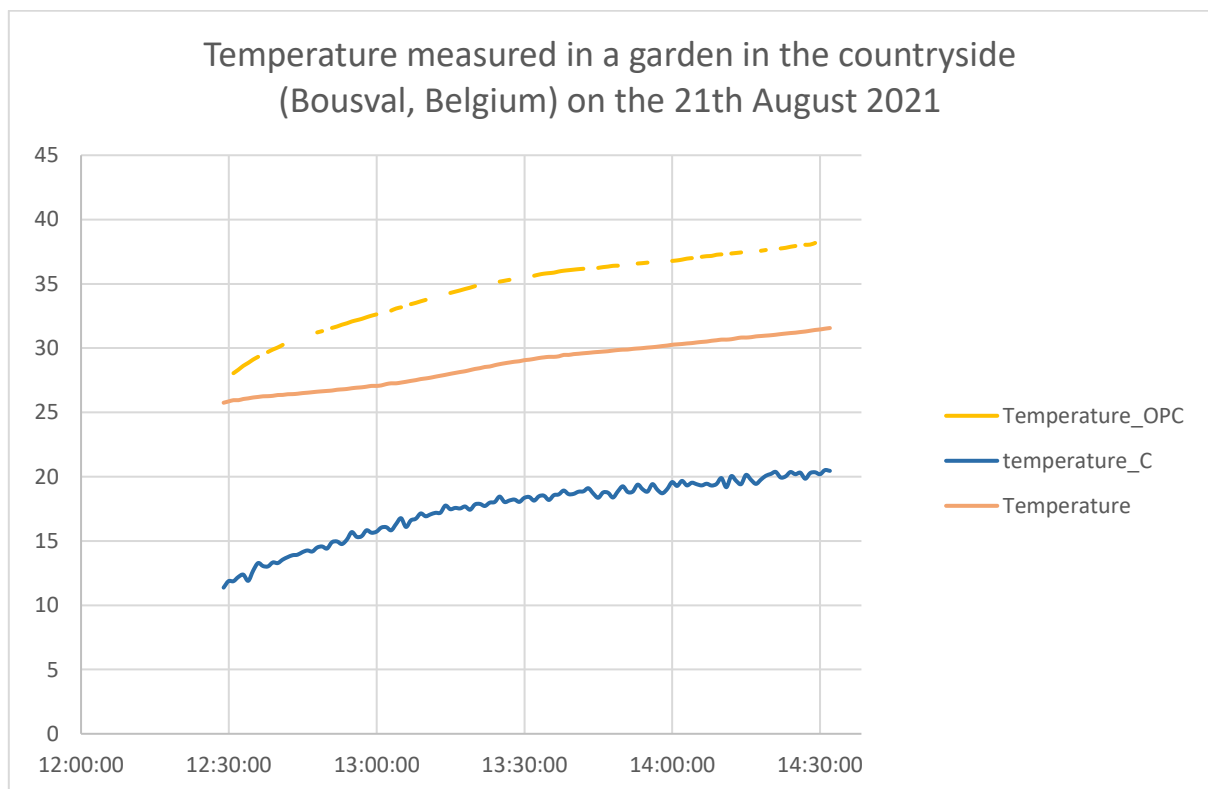


Figure 78 Graph of the temperature measured in a garden in the countryside

Source: own work, using the Seacanairy

2 Particulate matter

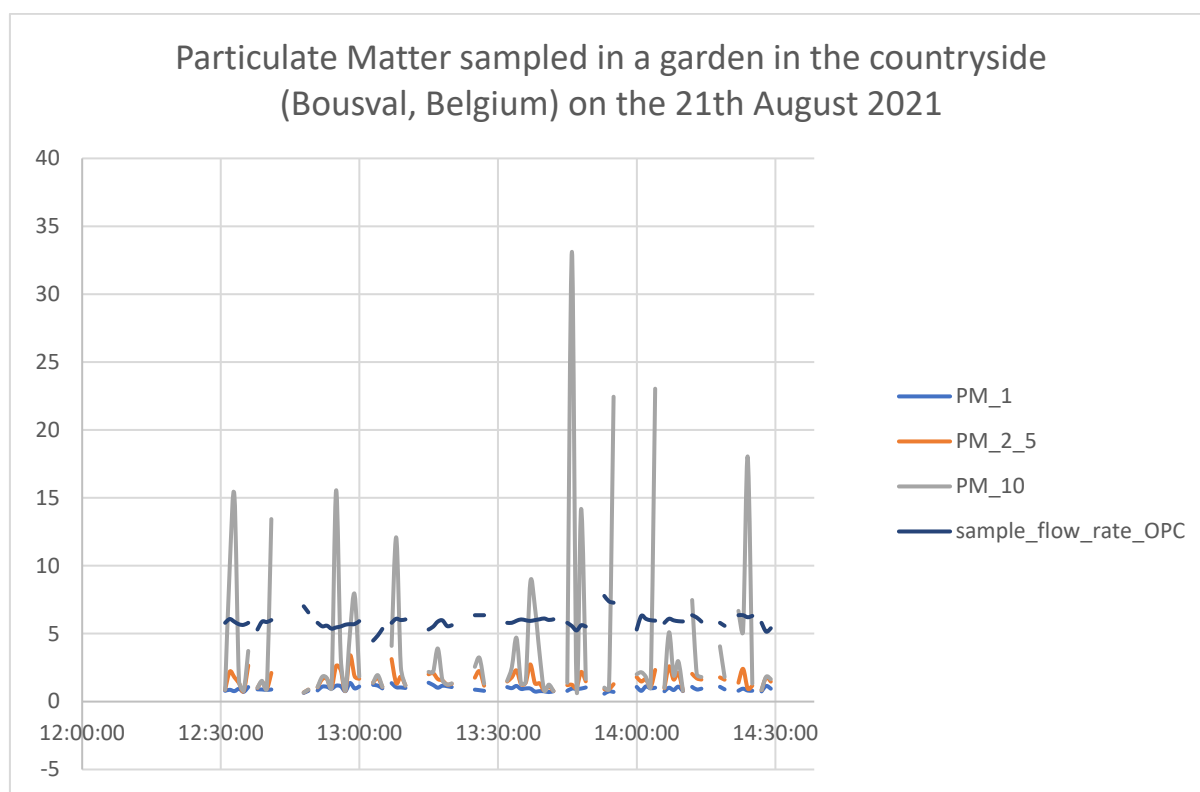


Figure 79 Graph of particulate matter sampled in a garden in the countryside

Source: own work, using the Seacanairy

3 Gas sensors

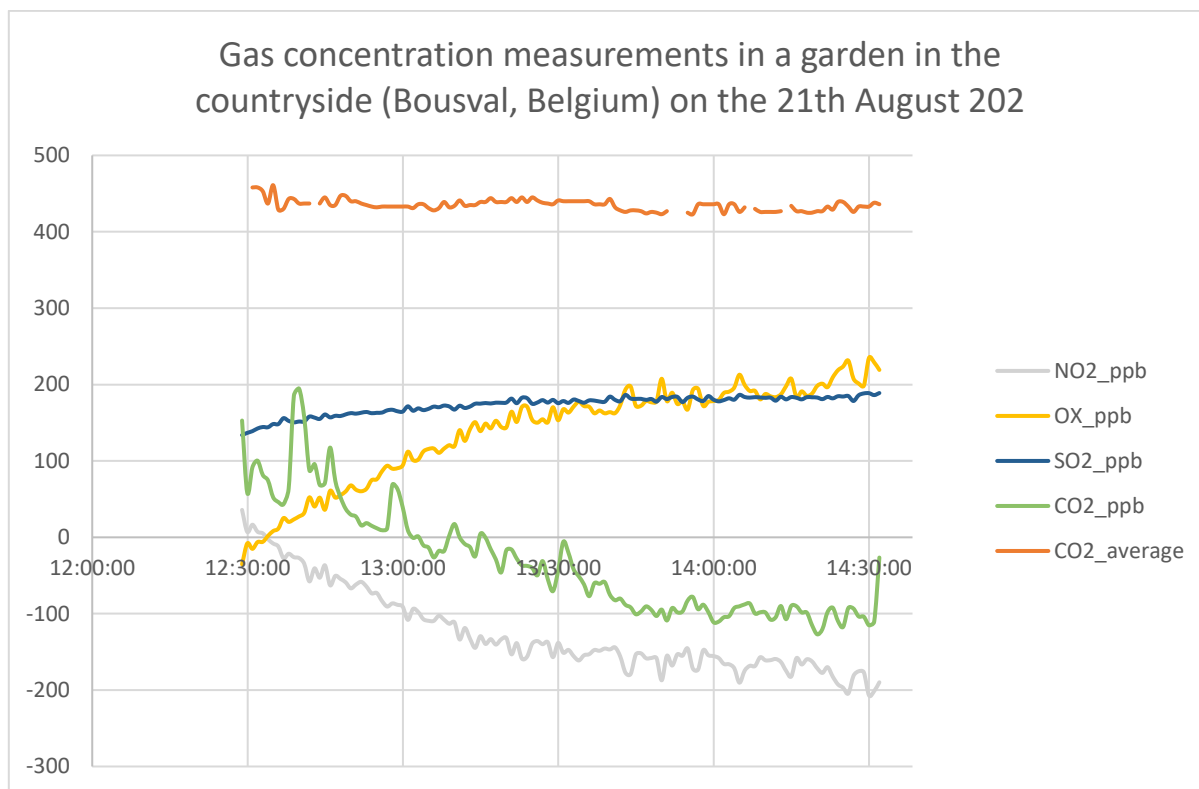


Figure 80 Graph of gas concentration in a garden in the countryside

Source: own work, using the Seacanairy

4 Air flow

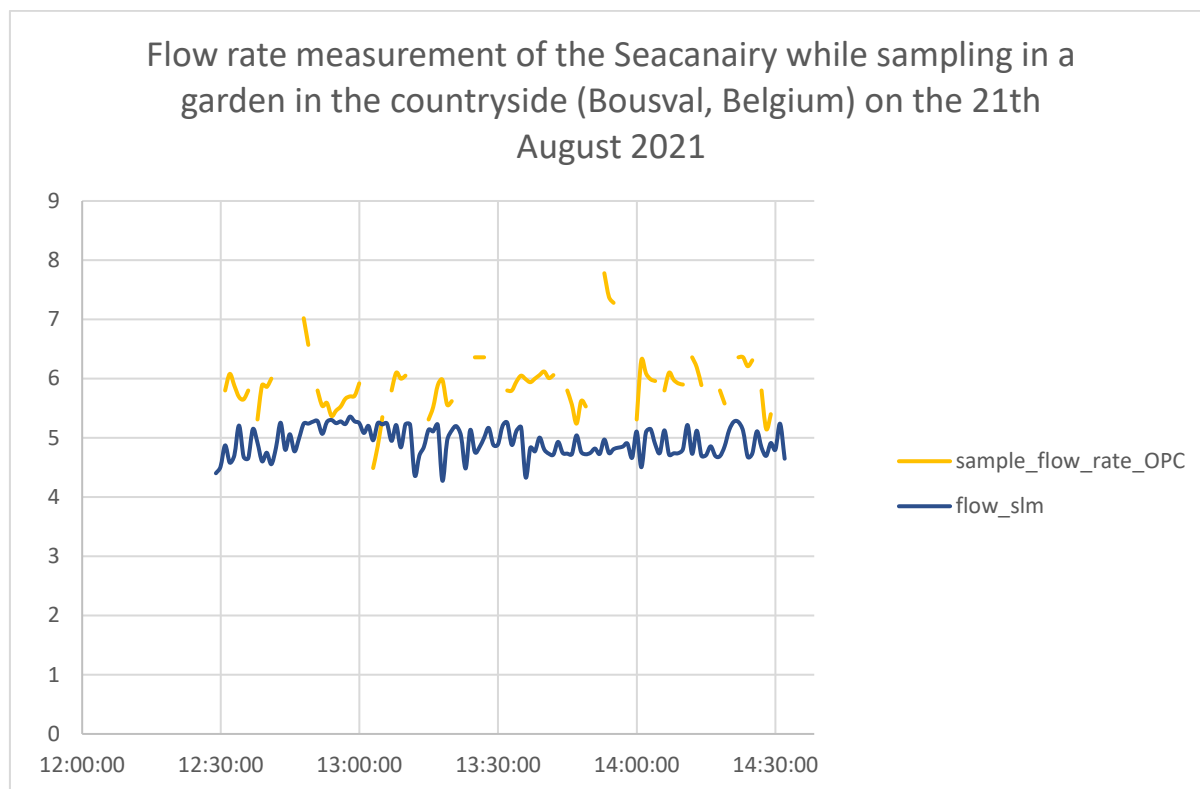


Figure 81 Graph of the flow rate measurement of the Seacanairy while sampling in a garden in the countryside

Source: own work, using the Seacanairy

5 SO2 peak when a nearby lawn tractor passes

The carbon oxides peak is generated by the passage of a lawn tractor nearby.

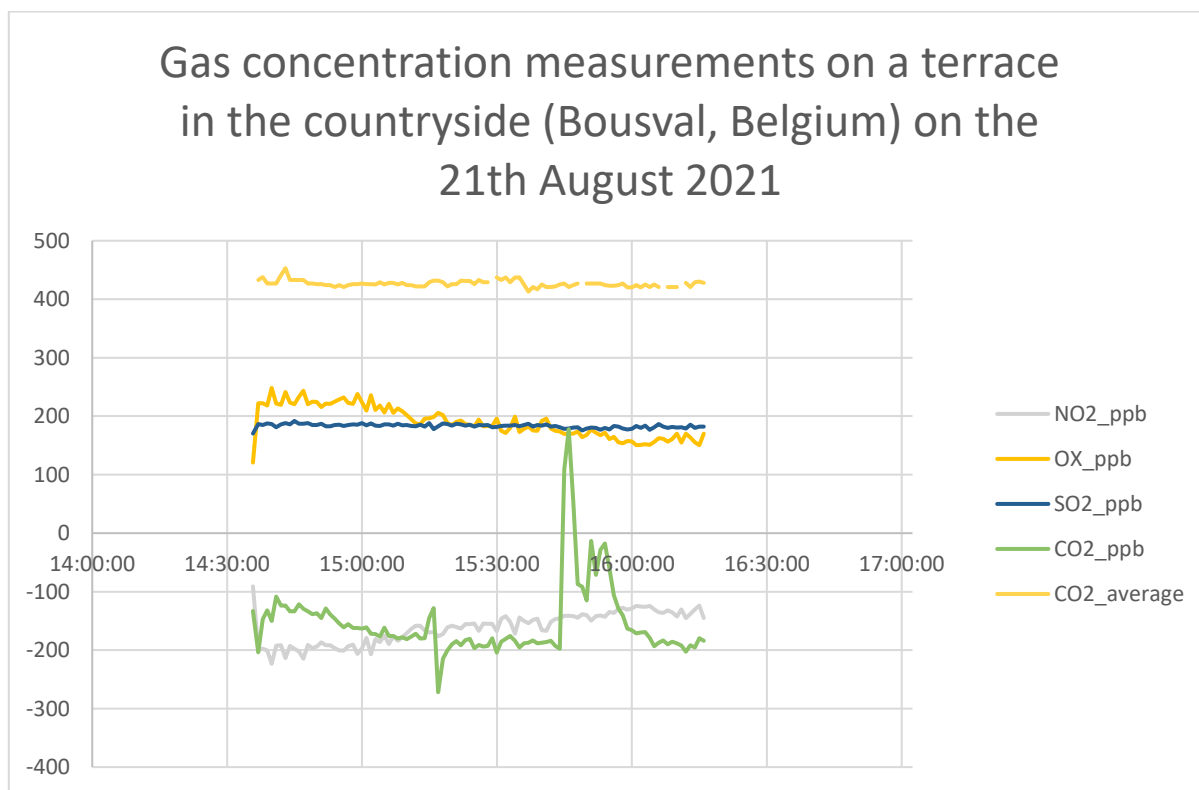


Figure 82 Graph of gas concentration measurements on a terrace in the countryside

Source: own work, using the Seacanairy

Bibliography

- [1] Alchemy Power Inc. (2020) 'Pi-16ADC for Raspberry Pi™'. <https://www.alchemy-power.com/wp-content/uploads/2020/02/16ADC20200203-1-DS.pdf>
- [2] Alphasense Ltd (2019) 'Alphasense User Manual OPC-N3 Optical Particle Counter'.
- [3] Alphasense Ltd and Mark Giles (2019) 'Supplemental SPI information for the OPC-N3'.
- [4] Cburnett (2006) *English: A single master and three slaves on a Serial Peripheral Interface (SPI) bus*. https://commons.wikimedia.org/wiki/File:SPI_three_slaves.svg (Accessed 17 May 2021).
- [5] 'Different Types Of Cable Lugs With PDF File' (2019) *Engineering Discoveries*. <https://engineeringdiscoveries.com/different-types-of-cable-lugs-with-pdf-file/> (Accessed 19 August 2021).
- [6] Dinh, T.-V., Choi, I.-Y., Son, Y.-S. and Kim, J.-C. (2016) 'A review on non-dispersive infrared gas sensors: Improvement of sensor detection limit and interference correction'. *Sensors and Actuators B: Chemical*, 231, pp. 529–538. doi:10.1016/j.snb.2016.03.040
- [7] Dowker, K. P. and Hardwick, K. (2008) 'Effect of tubing type on gas detector sampling systems (RR635)'. <https://www.hse.gov.uk/research/rrpdf/rr635.pdf>
- [8] E+E Elektronik (2020) 'Utilising the E2 Interface on EE894'. https://downloads.epluse.com/fileadmin/data/product/ee894/Utilising_E2_Interface_EE894_AN1808-1.pdf
- [9] E+E Elektronik Ges.m.b.H. (n.d.) 'CO2 Module EE894 Protocol Description I2C'. https://downloads.epluse.com/fileadmin/data/product/ee894/TUG_EE894_I2C.pdf
- [10] 'EE894 - CO2 Module Measures Four Climate Parameters' (n.d.) <https://www.epluse.com/en/products/co2-measurement/co2-sensor/ee894/>
- [11] 'EE894 datasheet' (n.d.) https://downloads.epluse.com/fileadmin/data/product/ee894/datasheet_EE894.pdf
- [12] 'FEP (fluorinated ethylene propylene) - Polyflour' (n.d.) <https://www.polyflour.nl/en/materials/fep/> (Accessed 1 May 2021).
- [13] Haider, A., Robert, M. and Schwarz, R. (n.d.) 'Specification E2 Interface'. http://downloads.epluse.com/fileadmin/data/sw/Specification_E2_Interface.pdf

-
- [14] Hinds, W. C. (1999) *Aerosol Technology: Properties, Behavior, and Measurement of Airborne Particles*. John Wiley & Sons.
- [15] 'T°C' (2021) *Wikipedia*.
<https://en.wikipedia.org/w/index.php?title=I%C2%B2C&oldid=1019929433>
(Accessed 22 May 2021).
- [16] 'Installing packages using pip and virtual environments – Python Packaging User Guide' (n.d.) <https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/> (Accessed 9 May 2021).
- [17] 'JavaScript Object Notation' (2021) *Wikipédia*.
https://fr.wikipedia.org/w/index.php?title=JavaScript_Object_Notation&oldid=179637222 (Accessed 13 May 2021).
- [18] Lindegaard, K.-P. (n.d.) 'smbus2 0.4.1 Documentation'.
<https://github.com/kplindegaard/smbus2> (Accessed 15 May 2021).
- [19] MATT (2018) 'Introducing the Raspberry Pi 3 B+ Single Board Computer'. *Raspberry Pi Spy*. <https://www.raspberrypi-spy.co.uk/2018/03/introducing-raspberry-pi-3-b-plus-computer/> (Accessed 10 May 2021).
- [20] M&C TechGroup Germany GmbH (n.d.) 'Instruction manual - Bellows pump series MP®-F'. https://www.mc-techgroup.com/manuals/M_MPF_EN.pdf
- [21] MOCQ, F. (2017) 'Le port série du Raspberry Pi 3 : pas simple !' *Framboise 314, le Raspberry Pi à la sauce française....* <https://www.framboise314.fr/le-port-serie-du-raspberry-pi-3-pas-simple/> (Accessed 25 May 2021).
- [22] MOCQ, F. (2019) 'Utiliser le port série du Raspberry Pi 3 et du Pi Zero'. *Framboise 314, le Raspberry Pi à la sauce française....* <https://www.framboise314.fr/utiliser-le-port-serie-du-raspberry-pi-3-et-du-pi-zero/> (Accessed 25 May 2021).
- [23] 'OPC-N3 Particle Monitor' (n.d.) <https://www.alphasense.com/WEB1213/wp-content/uploads/2019/03/OPC-N3.pdf>
- [24] 'Particulates | Alphasense' (2015) *Alphasense | The Sensor Technology Company*.
<https://www.alphasense.com/index.php/products/optical-particle-counter/> (Accessed 17 April 2021).
- [25] 'Peli Storm iM2720 Case Call 01902 324734 For Best Prices' (n.d.)
<https://www.waterproof-cases.co.uk/product/peli-storm-im2720-case/> (Accessed 21 August 2021).
- [26] Pol Cuvelier (2021) 'Evening spent searching for a solution concerning the OPC-N3 and the electric pump'.
- [27] 'PTFE (polytetrafluoroethylene) - Polyfluor' (n.d.)
https://www.polyfluor.nl/html/index.php?page_id=86&language_id=2 (Accessed 1 May 2021).

-
- [28] 'Raspberry Pi UART Communication using Python and C | Raspberry Pi' (n.d.) <https://www.electronicwings.com/raspberry-pi/raspberry-pi-uart-communication-using-python-and-c> (Accessed 25 May 2021).
- [29] Saint-Gobain (n.d.) 'Tygon products for electronics purpose'. <https://www.processsystems.saint-gobain.com/products/electronics> (Accessed 10 May 2021).
- [30] Shugar, G. J., Ballinger, J. T. and Dawkins, L. M. (1996) *Chemical technicians' ready reference handbook* 4th ed. New York: McGraw-Hill.
- [31] 'SMBus Protocol - kernel.org' (n.d.) *The Linux Kernel Archives*. <https://www.kernel.org/doc/Documentation/i2c/smbus-protocol> (Accessed 15 May 2021).
- [32] Sousan, S., Koehler, K., Hallett, L. and Peters, T. M. (2016) 'Evaluation of the Alphasense Optical Particle Counter (OPC-N2) and the Grimm Portable Aerosol Spectrometer (PAS-1.108)'. *Aerosol science and technology : the journal of the American Association for Aerosol Research*, 50(12), pp. 1352-1365. doi:10.1080/02786826.2016.1232859
- [33] 'Technical User Guide - Protocol Description I2C' (n.d.) https://downloads.epluse.com/fileadmin/data/product/ee894/TUG_EE894_I2C.pdf
- [34] Thoms, V. (n.d.) *spidev: Python bindings for Linux SPI access through spidev*. Python <http://github.com/doceme/py-spidev> (Accessed 23 May 2021).
- [35] 'UART configuration - Raspberry Pi Documentation' (n.d.) <https://www.raspberrypi.org/documentation/configuration/uart.md> (Accessed 8 April 2021).
- [36] 'Updating and upgrading Raspberry Pi OS - Raspberry Pi Documentation' (n.d.) <https://www.raspberrypi.org/documentation/raspbian/updating.md> (Accessed 9 May 2021).
- [37] Van der Borght, L. (2020) 'Constructie en kalibratie van een toestel voor het meten van de luchtkwaliteit aan boord van zeeschepen'. Antwerp Maritime Academy.