POLITECNICO DI TORINO

Master's Degree in Agritech Engineering

Master's Degree Thesis

From Load Cell to Cloud User Interface: End-to-End Engineering of a Modular, Multi-Channel Pot-Scale IoT Weighing Lysimeter

Supervisors

Prof. Umberto GARLANDO

Prof. Alessandro SANGINARIO

Ph.D. Mattia BAREZZI

Candidate

Filippo MIRALDI

October 2025

Summary

This thesis presents the design, implementation, and experimental validation of an IoT, high-precision, modular weighing lysimeter to be used inside a plant growth chamber.

Weighing lysimeters are used to precisely quantify plant water losses (evapotranspiration) and growth by monitoring the changes in the weight of the pots used to grow them.

The requirements called for a system capable of monitoring four pots concurrently and transmitting the data to a central server using a Raspberry Pi single-board computer.

The proposed architecture couples a commercial mechanical structure (CELMI LP7611 single-cell load frame) with two custom analog front-end alternatives, based respectively on the Nau7802 IC and the ZSSC3224 IC, which digitize the Wheatstone-bridge load-cell analog signals. Both proposals feature input filtering, dedicated regulated power for the analog section, and a PCA9615 differential I2C transceiver IC that reliably transmits the digitized data over Cat5 Ethernet cabling to the Pi_master custom Raspberry Pi shield. This shield, using a MAX7320 I2C port expander IC, multiplexes all the scale signals to a single Raspberry Pi I2C port. The Pi_master also features two STEMMA-QT ports, a reliable and proven standard that can be used to expand the capabilities of the system, thanks to the wide commercial availability of sensor modules and actuators featuring this interface.

The flexibility of the system was demonstrated when it was expanded to accommodate an additional irrigation control system based on a commercially sourced, inexpensive relay board. This board is used to control small pumps to selectively irrigate each pot. It was adapted using an MCP23017 I2C GPIO expander breakout board through the STEMMA-QT port.

On the software side, an edge stack written in Python and running on the Raspberry Pi handles sensor polling, calibration, channel selection, irrigation actuation, and MQTT publishing. A central IoT server, running Eclipse Mosquitto, Telegraf, InfluxDB, and Grafana, is installed in Docker on an Amazon Lightsail cloud Virtual Private Server. This stack ingests, stores, and visualizes the data

generated by each scale system.

Performance validation demonstrates stable and precise weight acquisition, robust communication, and ease of use, proving the excellent capabilities of the system, which rivals equivalent commercial solutions at a much higher cost. These results highlight its potential as an enabling tool for the testing and validation of wearable plant sensors and for developing irrigation algorithms.

Acknowledgements

"Alla mia mamma, al mio papà e a mio fratello Matteo che mi sostengono, supportano e sopportano in ogni progetto"

Vorrei esprimere la più sincera gratitudine a tutta la mia famiglia e ai miei amici che mi hanno accompagnato durante questi anni di Università e ci sono sempre stati in ogni momento. La sensazione più bella è sapere di avere qualcuno sempre al proprio fianco pronto a sostenerti in qualunque momento.

In particolare, vorrei ringraziare i miei nonni. Giovanna mi ha trasmesso la sua infinita energia, forza e determinazione; Giorgio ha stuzzicato in me la voglia di fare e mi ha insegnato che tutto può avere un valore e trasformarsi in qualunque forma, purché si disponga di sufficiente ingegno e fantasia; Nino mi ha insegnato l'importanza della precisione e della professionalità; Pina mi ha mostrato come la gentilezza e un piccolo gesto o pensiero possano essere importanti. Siete la mia guida.

Ringrazio il *Dipartimento di Elettronica e Telecomunicazioni* (DET), il *Dipartimento di Ingegneria dell'Ambiente, del Territorio e delle Infrastrutture* (DIATI) del Politecnico di Torino e tutte le *"Aziende Amiche"* per avermi offerto l'opportunità di sperimentare in vari ambiti e di apprendere un patrimonio di conoscenze che vanno oltre quello che si trova sui libri di testo.

Infine, esprimo gratitudine al Corso di *Laurea Magistrale in Agritech Engineering* per avermi dato l'opportunità di coniugare le mie due grandi passioni: l'agricoltura e l'elettronica. In particolare vorrei ringraziare tutto il gruppo *eLiONS* per la gentilezza, la professionalità e la cordialità che ogni singolo membro ha dimostrato durante tutto il mio percorso di tesi.

Table of Contents

List of Tables IX

List of Figures X

Acronyms XIV

1 Introduction 1
- 1.1 Presentation of the Issue . . . 1
- 1.2 System requirements . . . 3

2 General System Architecture 4
- 2.1 Scale . . . 5
  - 2.1.1 Load Cell . . . 5
  - 2.1.2 Sensor Front-End . . . 8
- 2.2 Sensor Data Aggregator . . . 17
  - 2.2.1 Raspberry Pi 4 model B . . . 17
  - 2.2.2 Pi_Master . . . 18
- 2.3 Optional Accessories Boards . . . 21
- 2.4 Cloud Server . . . 22
  - 2.4.1 Eclipse Mosquitto . . . 24
  - 2.4.2 InfluxData Telegraf . . . 24
  - 2.4.3 InfluxDb . . . 24
  - 2.4.4 Grafana . . . 25

3 Slave_NAU7802 Hardware Design 26
- 3.1 Schematics . . . 27
  - 3.1.1 Input Filter . . . 27
  - 3.1.2 NAU7802 - ADC . . . 30
  - 3.1.3 PCA9615 - Digital Data Transmission . . . 34
- 3.2 PCB Implementation . . . 37
  - 3.2.1 Electrical Design . . . 37

3.2.2 Mechanical Design 41

4 Slave_ZSSC3224 Hardware Design 44
4.1 Schematics 46
4.1.1 Input Filter 46
4.1.2 ZSSC3224 - ADC 47
4.1.3 3.3V Regulator 48
4.1.4 PCA9615 - Digital Data Transmission 50
4.2 PCB Implementation 51
4.2.1 Electrical Design 51
4.2.2 Mechanical Design 52

5 Pi_Master Hardware Design 53
5.1 Schematics 55
5.1.1 PCA9615 - Digital Data Transmission 55
5.1.2 Channel Multiplexer 56
5.1.3 Logic Level Shifter and Stemma QT 59
5.2 PCB implementation 61
5.2.1 Electrical Design 62
5.2.2 Mechanical Design 63

6 Optional Accessories Board Hardware 64

7 Edge Software 69
7.1 Introduction 69
7.2 OS Installation 70
7.3 Python Code 73
7.3.1 main.py 74
7.3.2 scales.py 80
7.3.3 channel_select.py 88
7.3.4 csv_conf.py and scale_system_calibration_and_data.csv . 89
7.3.5 i2c_bus.py 90
7.3.6 irrigation.py 91
7.3.7 mqtt_conf.py 93
7.4 Python-OS Interaction and Software Setup 94

8 Server 98
8.1 Amazon Lightsail VPS 99
8.2 Containerization and Docker 100
8.2.1 Docker Installation and Important Docker characteristics . . 102
8.3 Portainer 105
8.3.1 Installation 105

8.3.2 Features . . . 107
8.4 IoT Stack . . . 110
8.4.1 Eclipse Mosquitto . . . 110
8.4.2 InfluxData Telegraf . . . 118
8.4.3 InfluxData InfluxDB . . . 124
8.4.4 Grafana . . . 127

9 Performance evaluation and validation 132
9.1 Experimental Setup . . . 133
9.2 Experimental Procedure . . . 134
9.3 Results . . . 136

10 Final considerations 140
10.1 Economic considerations . . . 140
10.1.1 First competitor: industrial automation approach . . . 141
10.1.2 Second competitor: fully commercial solution . . . 143
10.1.3 Comparison of the four solutions . . . 144
10.2 Future improvements . . . 147
10.2.1 Scale system . . . 147
10.2.2 Irrigation board . . . 148

11 Conclusion 151

A ZSSC3224 adress definition code 153

B MQTT configuration messages examples 154

C Usage of python-scales-installer.sh 155

Bibliography 156

List of Tables

2.1 Comparison of output characteristics at different voltages. 9
2.2 Possible PGA gain summary 14

3.1 NAU7802 regulator voltage settings 32
3.2 Recommended C7 values for different voltages 32

5.1 MAX7320 Address Map 58

9.1 Analog front-ends theoretical input characteristics with the 50Kg load cell using their typical excitation voltages. 134

10.1 Costs (excluding VAT) of the alternative solution using the industrial automation approach and the fully commercial solution 145
10.2 Costs (excluding VAT) of the custom solution using Slave_nau7802 and Slave_nau7802 front ends 146

List of Figures

1.1 *eLiONS lab* wearable plant sensor [1] 1

2.1 General block diagram 4
2.2 Overview and internal structure view of *LP7611 single cell*. 5
2.3 Strain gauge [6] 6
2.4 Wheatstone bridge 7
2.5 Single strain gauge load cell [7] 7
2.6 Full bridge Type 1 load cell [7] 8
2.7 *Slave_nau7802* assembly 10
2.8 *NAU7802* block diagram [9] 11
2.9 Delta-sigma modulator [11] 11
2.10 *Slave_zssc3224* assembly 13
2.11 *ZSSC3224* block diagram [13] 14
2.12 I2C signal diagram [14] 15
2.13 *PCA9615* block diagram [15] 16
2.14 *Raspberry Pi 4* with the custom shield *Pi_master* 17
2.15 Raspberry Pi 4 model B 18
2.16 *Pi_master* assembly 19
2.17 *MAX7320* block diagram [18] 20
2.18 *MAX7320* output port structure 20
2.19 Relay expansion board 21
2.20 *MCP23017* block diagram [19] 22
2.21 Cloud architecture block diagram 23
2.22 *Eclipse Mosquitto* logo [21] 24

3.1 General block diagram - NAU7802 scale subsystem 26
3.2 Input filter schematic diagram *NAU7802* 27
3.3 Relative capacitance change on two 0.01uF MLCC under applied DC bias voltage [23]. 29
3.4 NAU7802 subsection setup for 4.5V bridge supply 30
3.5 Load cell connection diagram 31

3.6 Digital interface section schematic 34
3.7 Output results of *Saturn PCB Design Toolkit Version 8.41* 35
3.8 Side by side view of the project file and the final PCB manufactured 37
3.9 *JLCPCB JLC04161H-7628* stackup 38
3.10 Scale hole square pattern 41
3.11 Board mounted underneath the scale 42
3.12 Views of the rivet nut and mounting tool used in the project 43

4.1 General block diagram - ZSSC3224 scale subsystem 44
4.2 Input filter schematic diagram ZSSC3224 46
4.3 ZSSC3224 subsection circuit diagram 47
4.4 3.3V Regulator subsection circuit diagram 48
4.5 Digital interface and logic lever shifter section schematic 50
4.6 Side by side view of the project file and the final PCB manufactured 51
4.7 Side-by-side view of the two analog sensor front ends 52

5.1 General block diagram - sensor data aggregator subsystem 53
5.2 One channel schematic of the digital interface section 55
5.3 Channel multiplexer circuit diagram 56
5.4 Logic level shifter and STEMMA QT ports circuit diagram 59
5.5 Side by side view of the project file and the final PCB manufactured 61
5.6 Pi_master mounted on the Raspberry Pi 63

6.1 Small submersible pump used [29] 65
6.2 Relay board to control pumps 66
6.3 Relay board input section 66
6.4 Adafruit 5346 I/O expander [30] 67
6.5 Adafruit 5346 circuit diagram [31] 67
6.6 MCP23017 I2C adress byte depending on A0,A1 and A2 value . . . 68
6.7 Assembled relay expansion board 68

7.1 *Raspberry Pi Imager* first page view 70
7.2 OS customization banner 71
7.3 Customization menu 72

8.1 Server system architecture block diagram 98
8.2 Amazon Lightsail Logo 99
8.3 Comparison between container-based and VM-based architecture. . 101
8.4 Portainer environment selection 107
8.5 Portainer dashboard 108
8.6 Portainer container page 108
8.7 Detail on Quick Actions from the "container" page 109

8.8 State control buttons 109
8.9 Eclipse Mosquitto logo 110
8.10 Top subsection in new Docker container creation page 111
8.11 Top sub-section of Mosquitto configuration 111
8.12 Commands and logging settings 112
8.13 Volumes and Bindings 112
8.14 Mosquitto binds 113
8.15 Network settings 113
8.16 Environment variables 114
8.17 Restart policy options 114
8.18 Runtime and resource constraints 115
8.19 Container console page 117
8.20 InfluxData Telegraf logo 118
8.21 Volumes configuration page 119
8.22 Network configuration page 119
8.23 InfluxDB data structure 124
8.24 API token page of InfluxDB 126
8.25 Logo Grafana 127
8.26 Grafana data sources page 128
8.27 Grafana data source configuration 129
8.28 Grafana new dashboard page 130
8.29 Grafana visualization creation page 131

9.1 Performance evaluation and validation test setup block diagram . . 133
9.2 Calibration weights set 135
9.3 Histogram of readings using Slave_nau7802 front ends 136
9.4 Comparison of raw Slave_nau7802 data and the same data processed with a moving average filter. 137
9.5 Histogram of readings using Slave_zssc3224 front ends 138
9.6 Comparison of raw Slave_zssc3224 data and the same data processed with a moving average filter. 139

10.1 Celmi Data load cell interface [33] 141
10.2 Teltonika RUT906 IoT gateway [34] 142
10.3 Kern & Sohn model KGP 60K-3 IoT scale [35] 143
10.4 Adafruit I2C to 8 Channel Solenoid Driver [36] 148
10.5 Comparison of pump wiring diagram using the two different boards 150

Acronyms

ADC

Analog to Digital Converter

API

Application Programming Interface

ARM

Advanced RISC Machine

CLI

Command Line Interface

CPU

Central Processing Unit

DAC

Digital to Analog Converter

DHCP

Dynamic Host Configuration Protocol

DNS

Domain Name System

DSP

Digital Signal Processing

GPIO

General-Purpose Input/Output

GUI

Graphical User Interface

I2C

Inter-Integrated Circuit

IC

Integrated Circuit

IoT

Internet of Things

JSON

JavaScript Object Notation

LAN

Local Area Network

LDO

Low-dropout regulator

LED

Light-Emitting Diode

LoRa

Long Range protocol - physical layer

MLCC

Multilayer Ceramic Capacitors

MQTT

Message Queuing Telemetry Transport

OS

Operating System

PCB

Printed Circuit Board

PGA

Programmable Gain Amplifier

PoliTO

Polytechnic University of Turin

QoS

Quality of Service

RAM

Random Access Memory

SBC

Single Board Computer

SMD

Surface-Mount Device

VM

Virtual Machine

VPS

Virtual Private Server

Chapter 1

Introduction

1.1 Presentation of the Issue

Plants are incredible organisms that might appear passive and unfeeling, but they are far from it. Much like animals, they experience stress in different ways, ranging from water shortage, pathogens, environmental factors, and nutrient deficiencies. Among these, water stress is the most common. The issue is that, differently from animals, when the first signs of stress appear, it is already too late, and the quality and quantity of yield are already compromised. It is essential to be able to find a way to anticipate stress symptoms and intervene to prevent the damage, a task that is impossible without using the latest technologies available.

Figure 1.1: *eLiONS lab* wearable plant sensor [1]

Wearable plant sensors emerge as a compelling solution to the above-mentioned problem since they offer the possibility to effectively and continuously monitor the biological activity of each plant. Even the slightest anomaly can trigger a targeted and immediate intervention by the farmer. It is for this reason that the *World Economic Forum* in 2023 inserted *wearable plant sensors* in their report *"Top 10 Emerging Technologies of 2023"*[2].

In this novel and innovative scenario, *eLiONS lab at PoliTO* is developing and refining an innovative LoRa connected *wearable plant sensor* to real-time monitor stem electrical resistance that is proving to be an excellent indicator of plant stress condition[3].

To develop an intelligent algorithm and improved hardware revisions, it is essential to be able to control the amount of stress that the plant is facing, and thus check if the response of the system is what is expected.

To minimize the stress, besides optimal agronomic practices, plants are grown in a growth chamber where all environmental parameters are controlled to simulate various climates and meteorological conditions precisely, but the issue of water stress remains unsolved.

Water is lost by plants in a phenomenon called evapotranspiration. As the name implies, evapotranspiration is governed by evaporation and transpiration. The evaporation process is the transfer of water from the soil to the atmosphere, while transpiration is the transfer of water from the plant to the atmosphere. This water has to be replenished and kept at the optimal level; otherwise, the plants will enter a stress condition.

There exist various methodologies and instruments used to calculate evapotranspiration, but the most direct and dependable way is the *Lysimeter*[4]. Etymologically speaking, the word *Lysimeter* derives from the Greek *"lysis"*, meaning movement, and *"metron"*, meaning to measure. In particular, a *Weighing Lysimeter* is an instrument used to monitor the weight changes over time of a soil sample. These changes are directly translatable to soil water loss, because of percolation and evapotranspiration, or gain, thanks to irrigation or rain.

Weighting Lysimeters are containers filled with soil where crops are grown. Generally, these thanks are kept buried in the ground to measure the percolation of water in an open field. The lysimeter must be big enough to limit the boundary effects in plant root development, a minimum of 1 m^3 for small crops [4]. Additionally, the soil and the plants in the *lysimeter* must be kept with the same characteristics and conditions as the rest of the field. This kind of lysimeter is particularly interesting for the analysis of wearable plant sensors since it is fast, compared to the alternatives. It can measure evapotranspiration for periods as short as 10 minutes, giving precious insights into evapotranspiration patterns and variability during the day and night.[4]

Weighting Lysimeter are generally of two kinds [5]:

- ***platform lysimeter***: a single-point load cell is placed between two plates forming a traditional scale. The soil container is placed directly on top. This system is potentially problematic in field application because bigger soil containers need bigger platforms, increasing complexity and costs. Additionally, the sensing element is placed deep underground.

- ***suspension lysimeter***: The soil container is suspended using a rope from an above-ground structure. An s-beam load cell is used to monitor the tension on the rope and thus the weight. With this solution, no elements need to be placed underground, but there is the added complexity of the above-ground structure.

Since the plants at *eLiONS lab* are kept in a vase in growth chambers, the issue simplifies. With careful management of irrigation water loss through percolation is eliminated, and there is no need to place a container underground, so a platform lysimeter is adequate and reduces complexity.

1.2 System requirements

The issue calls for the development of a system capable of monitoring the weight of up to four pots, the maximum number that fits in a single growth chamber. Multiple systems will be built for multiple growth chambers. It must be based on a *Raspberry Pi* SBC and provide an adequate resolution and precision to monitor changes in the weight of pots under 5L in volume.

The data has to be safely stored and easily accessible for further analysis.

Modularity is key, and the system must incorporate the irrigation control hardware, too.

Chapter 2

General System Architecture

This chapter is devoted to explaining the structure of the proposed scale system and its main electronic components.

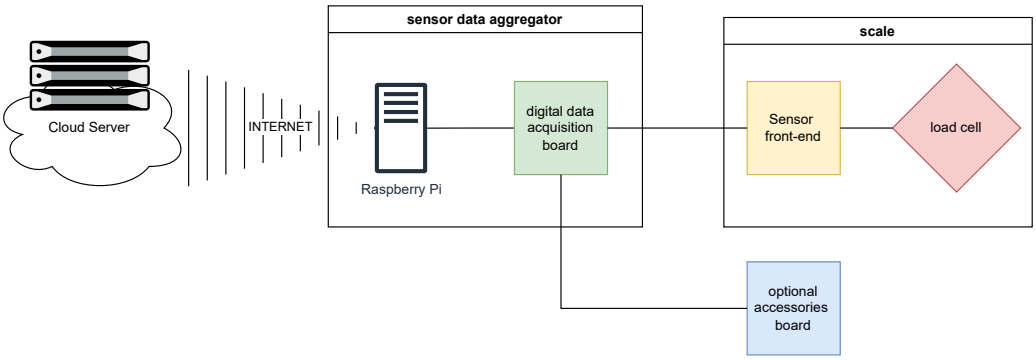
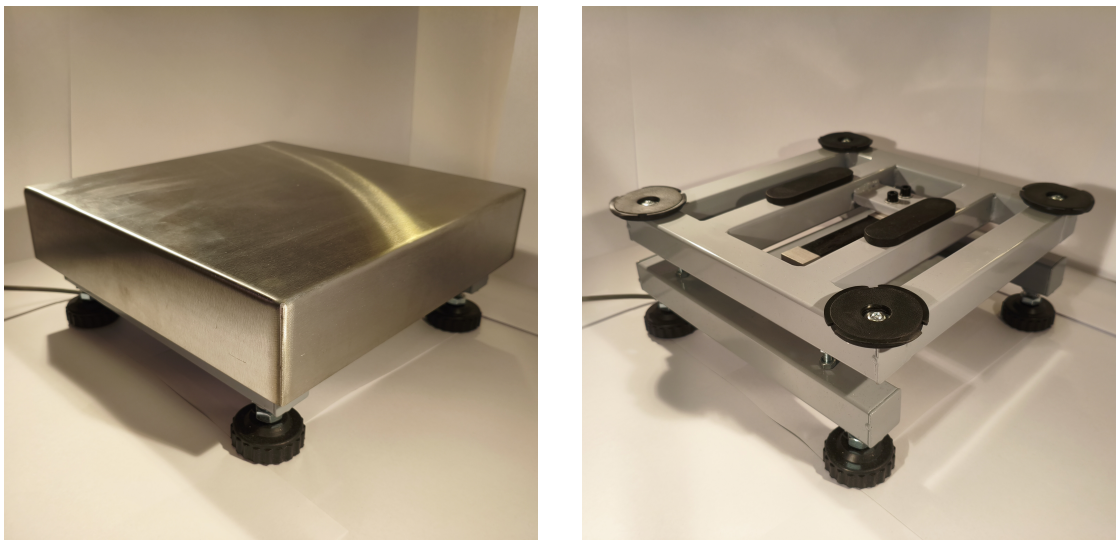


Figure 2.1: General block diagram

Figure 2.1 illustrates the block diagram comprising the six main components of the system. Starting from the right, it is present the scale subsystem, consisting of a load cell and a custom sensor front end; the sensor data aggregator, composed of a Raspberry Pi SBC with a custom digital data acquisition board; an optional additional accessories board, and the cloud server that manages the entire system. One of these accessory expansion boards is used to control the irrigation.

2.1 Scale



(a) External view **(b)** Internal structure

Figure 2.2: Overview and internal structure view of *LP7611 single cell*.

The scale is based on a commercially sourced load cell mounted in a heavy-duty frame, *CELMI LP7611 single cell*. As shown in Figure 2.2, it features a sturdy and corrosion-resistant powder-coated steel frame composed of separate and independent lower and upper sections connected by the load cell, all protected by a stainless steel cover that doubles as the load-bearing surface of the scale.

2.1.1 Load Cell

To explain the working principle of a load cell, it is necessary first to understand the strain gauge. A strain gauge is a sensor whose resistance varies proportionally to the strain applied. Strain is the amount of deformation that a certain material experiences when a force is applied to it. The most widespread typology of strain gauge is the bonded metallic strain gauge (Figure 2.3), where a thin metal foil is laid on a flexible substrate in a zigzag pattern to maximize the amount of metal foil subject to strain. Strain causes the wire to stretch, thus getting momentarily longer and thinner. Its electrical resistance changes because of this physical deformation. Resistance is defined in Equation 2.1. Resistivity is a property of the material, and it is not changed by tiny physical deformations. On the other hand, strain slightly changes the length and the cross-sectional area of the conductor on the strain gauges, leading to a change in resistance, in a strain gauge, of a few hundred

milliohms.

$$R = \rho \cdot \frac{L}{A} \tag{2.1}$$

R: resistance (Ω), ρ: resistivity ($\Omega \cdot$ m), L: conductor length (m), A: cross-sectional area (m^2)

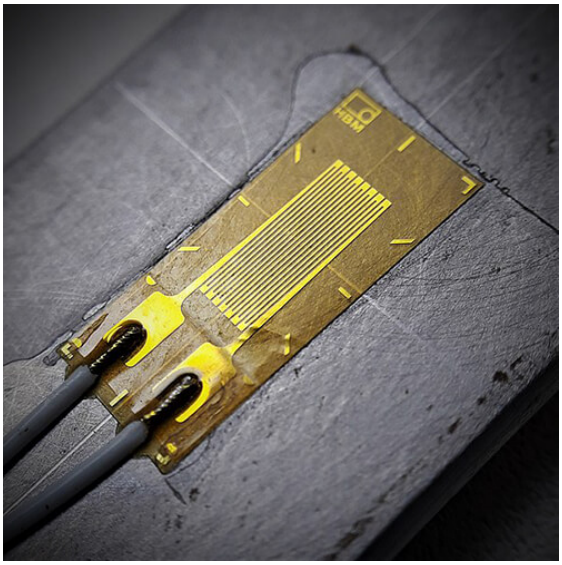


Figure 2.3: Strain gauge [6]

To measure small resistance changes, a Wheatstone bridge is commonly used. This is a circuit composed of two sets of two series resistors wired in parallel (figure 2.4). When all the resistors are equal, the voltage read between the two center nodes is 0V. If not all the resistors are equal, the bridge is said to be unbalanced, and now it is possible to read a voltage related to the relative value of the resistors and expressed by equation 2.2.

$$V_{out} = \left[\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right] \times V_{exc} \tag{2.2}$$

The simplest solution to read a strain gauge is to replace one of the fixed resistors in the bridge with the strain gauge, creating what is referred to as a quarter-bridge configuration.
To obtain a functional weighing measuring device, this single load cell, which on its own is very delicate, is stuck to a specifically characterized metal piece (figure 2.5). The well-defined deformation of the metal under load causes strain on the strain gauge, resulting in a resistance change. What has just been described is a crude implementation of a single-point load cell, but not an ideal one since it is

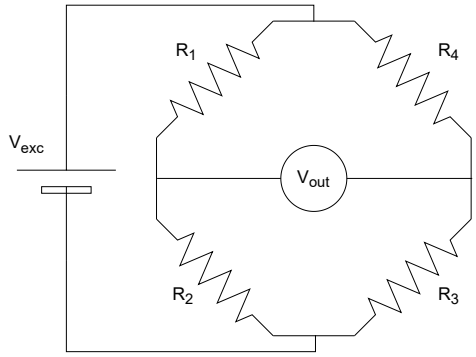


Figure 2.4: Wheatstone bridge

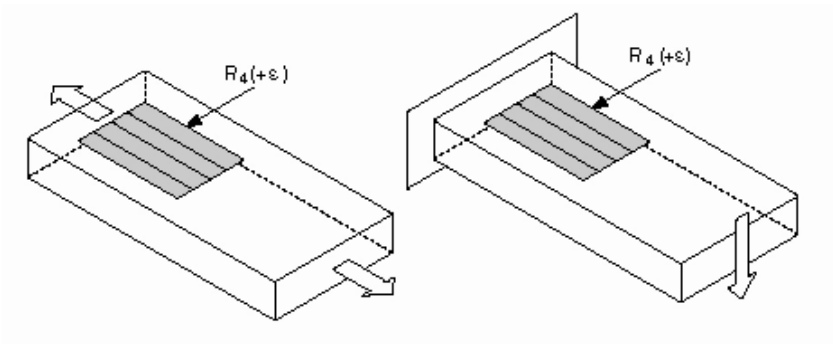


Figure 2.5: Single strain gauge load cell [7]

characterized by a lower sensitivity compared to other alternatives, it measures both axial and bending forces causing errors, since just the bending force is of interest in this weighting application, and it is sensible to errors related to temperature resistance drift in the strain gauges materials and in the external resistors needed to form the bridge [8].

A better solution to create a single-point load cell, also used in the commercial product chosen for this master's thesis, is the substitution of all four resistors in the bridge with strain gauges. Specifically, for a single-point load cell, the full-bridge type I configuration is of particular interest (figure 2.6). This configuration uses four strain gauges placed with the same orientation: two on top, subject to tensile strain, and two on the bottom, subject to compressive strain. Figure 2.6, with reference to Figure 2.4 for numbering, shows that the strain gauges are wired so that each branch of the Wheatstone bridge contains one gauge under tensile strain and one under compressive strain. Additionally, gauges on opposite sides of the bridge experience the same type of strain. The advantages of this configuration include:

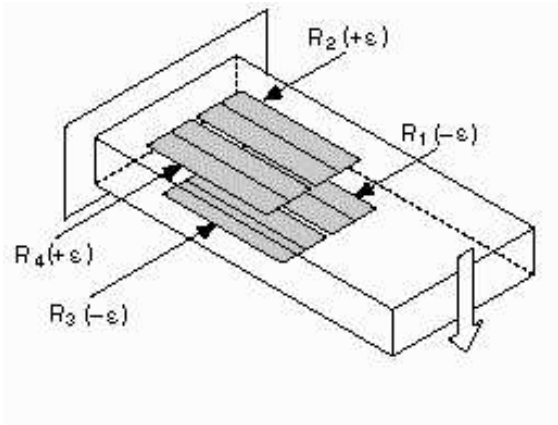


Figure 2.6: Full bridge Type 1 load cell [7]

- four times greater sensitivity compared to the quarter-bridge configuration,
- rejection of axial strain
- automatic compensation for temperature effects [8].

The load cell used is the *Zemic* model *L6N* with a maximum rated load of 10 kg.

2.1.2 Sensor Front-End

As explained in subsection 2.1.1, resistance changes in strain gauges are minute, and thus also the voltage output of a Wheatstone bridge is generally small, making a load cell a complex sensor to measure accurately. In this particular load cell, the full scale output, so the voltage read at the central nodes of the Wheatstone bridge with the maximum load applied on the cell representing the maximum output voltage, is given by equation 2.3. Table 2.1 shows what are the full-scale differential output and the variation in the output for 10 g resolution step. The output voltage is a differential voltage between the central node of the Wheatstone bridge. The voltage at each node with reference to ground is of $V_{\text{exc}}/2$ with no load on the scale, an offset several tens of times greater than the full-scale differential output, making it complex, if not impossible, to perform an accurate measurement calculating the difference of two single ended ADC readings without additional analog signal preparation. Working with such low-level signals, noise becomes an issue of primary importance. It is therefore best to keep the signal fully differential with differential ADC, differential filters, and differential amplifiers. Using a differential circuit to digitize only the voltage difference between the two central nodes of the Wheatstone bridge offers several key advantages: it automatically eliminates any common offset voltage in V_{out}, drastically improves noise immunity by rejecting common-mode

noise present on both branches of the signal, and significantly reduces power-supply noise contribution.

$$V_{out} = 0.002 \times V_{exc} \tag{2.3}$$

Bridge excitation:	**4.5V**	**3.3V**	**1.6V**
Full Scale Output	9mV	6.6mV	3.2mV
Δ V_{out} for 10g resolution step	9μV	6.6μV	3.2μV

Table 2.1: Comparison of output characteristics at different voltages.

The analog section of the sensor front end is the most crucial part to guarantee good system performance. For this reason, two different designs have been created and built to evaluate which one performs best experimentally. Both options are based on a dedicated IC for reading resistive sensors. The first solution is more economical and will be referred to as *slave_nau7802*, while the second one is more expensive and will be referred to as *slave_zssc3224*.

Slave_NAU7802

Slave_nau7802 is built around the *Nuvoton NAU7802* IC (figure 2.7).

Nau7802 is a highly integrated bridge sensor analog front end with a 24bit ADC with integrated PGA [9].

Figure 2.8 illustrates a simplified block diagram of the IC. The central nodes of the load cell are connected to pin VIN1P and VIN1N or to pin VIN2P and VIN2N. The input signal then encounters the first functional block, a differential multiplexer. It is independently controlled through the I2C interface and selects the signal that is sent to the rest of the internal circuits between one of the two input channels or the integrated analog temperature sensor. This temperature sensor outputs 109 mV at 25 °C with a sensor delta coefficient of 360 μV/°C. This sensor is used to compensate for temperature-related drift in sensors, and it is not used in this application since the load cell *Zemic L6N* is internally temperature compensated [10] using the methodology explained in subsection 2.1.1. The selected signal enters a differential PGA. The amplification can again be set through the I2C interface from 2 to 128. The PGA can also be bypassed if not necessary, with advantages in noise performance. The signal is now ready to enter the differential ADC. This is a 24-bit Delta sigma ADC.

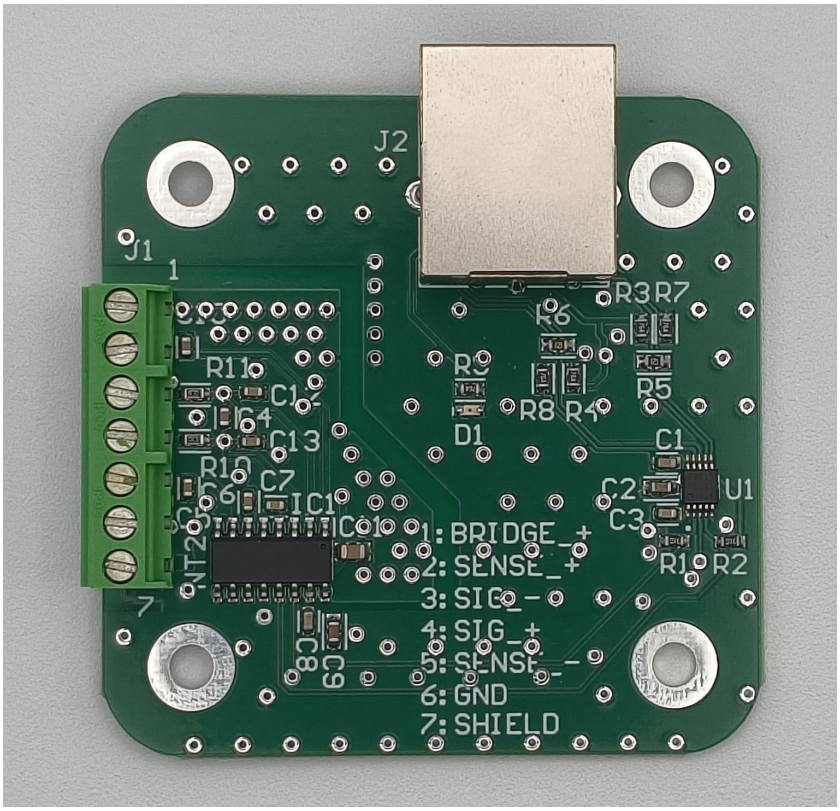


Figure 2.7: *Slave_nau7802* assembly

The working principle of this technology of ADC is explained in its essence in *MT022* from *Analog Devices* [11], and it is best summarized with an example. There are two main functional blocks: A Delta-sigma modulator and a digital signal processor. Figure 2.9 shows the basic block diagram of the simplest form of this kind of modulator. It is composed of a summing node, an integrator, a comparator, and a 1-bit DAC. The analog input signal (Vin), which is assumed to be 1V for this example, enters the summing node. From this input signal, B is subtracted. This voltage is considered to be +2.5V when the device is turned on; more details will be provided on this choice. The resulting -1.5V is fed to an integrator that calculates, essentially, the average of the signal. Since the system has just been started, the average of a single value is the value itself; the output of the integrator (A) is now the same as the input, so -1.5V. The signal at the input of the comparator is now negative, and thus its output is set to 0V, representing a digital 0 that is fed to a 1-bit DAC. The output of this DAC is -Vref if the input is 0 or +Vref if the input is 1. For this example, Vref is 2.5V. Since the input is 0 now, the B signal is -2.5V, which, when subtracted from the 1.5V input, makes the input of the comparator equal to 3.5V, which, averaged with the previous signal

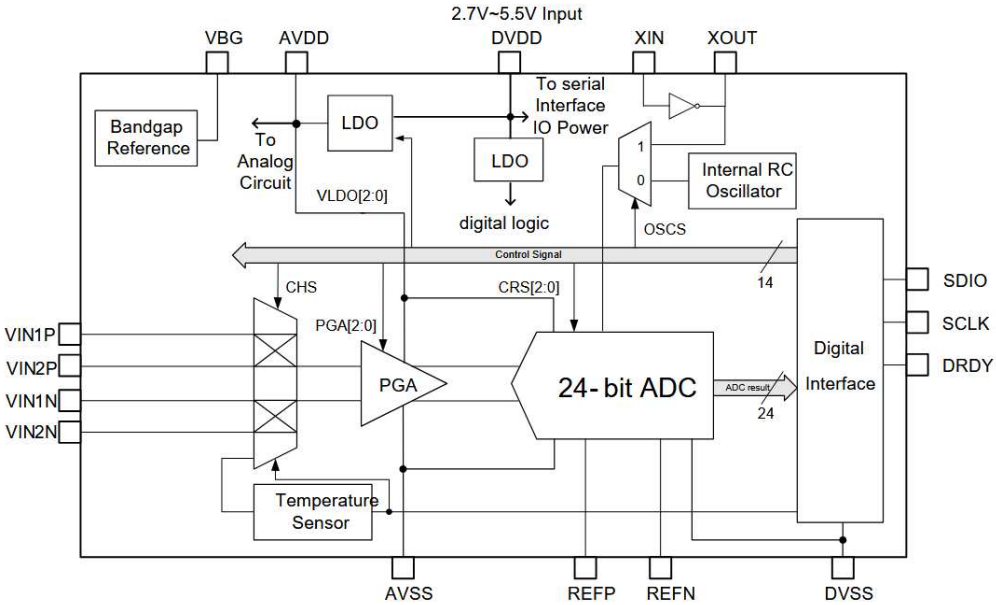


Figure 2.8: *NAU7802* block diagram [9]

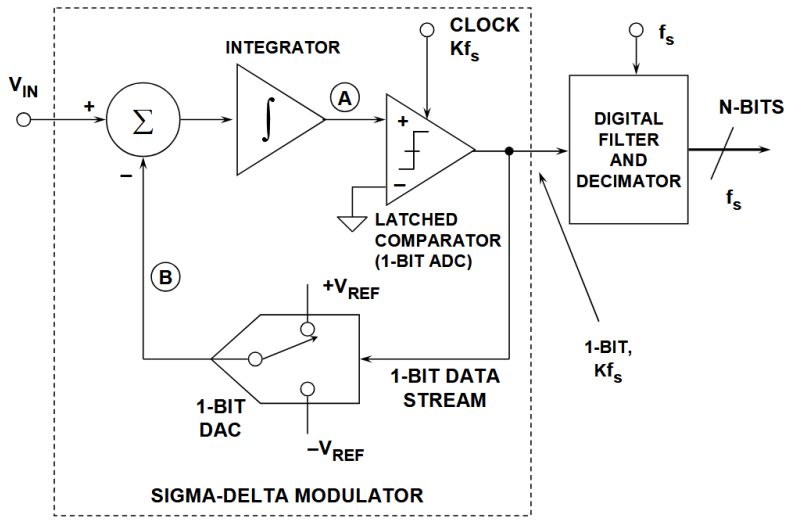


Figure 2.9: Delta-sigma modulator [11]

results in A equals to 1V. This results in an output from the comparator of 1, and then the circle repeats indefinitely. The output of the comparator can be recorded. It is a string of 0 and 1. From this string, it is possible to obtain the value of the

input signal since the concentration of 1 over the total of 1 and 0 is approximately equal to Vin over the reference voltage (in this case, 2Vref since it is a positive and negative reference voltage). The analog section of the ADC works at a way higher frequency compared to the frequency at which the digital results are outputted, since it is necessary to have a long binary string to properly compute the 1 density and obtain a good digital result. This last process that has been conceptually described is implemented by a digital low-pass filter combined with a decimator, which down-samples the input digital signal to an appropriate number of samples, creating the final 24-bit result.

The operation of a delta-sigma ADC is, thus, based on three main fundamental techniques that improve its performance [12]:

1. **Oversampling**: The input signal is sampled faster than the required Nyquist minimum, giving more data to the ADC to mitigate noise through averaging. Oversampling is inherent in the architecture of the ADC. The delta-sigma modulator has to work at a very high frequency, and its output is basically a fast stream of low-resolution samples that is then averaged to obtain the final output.

2. **Noise Shaping**: Thanks to the complex construction of this ADC, the quantization error is looped back into the input through the integrator. This distorts the noise frequency spectrum with a reduction of noise power in the lower band and a consequent increase in the upper bands, which are not of interest.

3. **Decimation**: Since the last digital section of the ADC is a digital low-pass filter, and noise shaping has pushed the majority of the noise power to higher frequencies, a significant portion of the noise power gets eliminated

Oversampling, noise shaping, and decimation give this kind of ADC high resolution with low noise, essential for high precision sensor measurement, at the cost of lower measurement speed, unimportant in this application, and complexity

NAU7802 integrates also a band gap reference voltage, an oscillator tuned at 4.9152 MHz ±3% for the ADC clock, and a software tunable LDO to power an external bridge sensor with a dedicated, precise, and low noise supply. Its output is settable to 2.4V, 2.7V, 3V, 3.3V, 3.6v, 3.9v, 4,2v and 4,5V.

Slave_ZSSC3224

Slave_zssc3224 is built around *Renesas ZSSC3224* IC (figure 2.10).

ZSSC3224 is a sensor signal conditioner IC built for the accurate amplification and digitization of differential input signals, in particular for bridge resistive sensors.

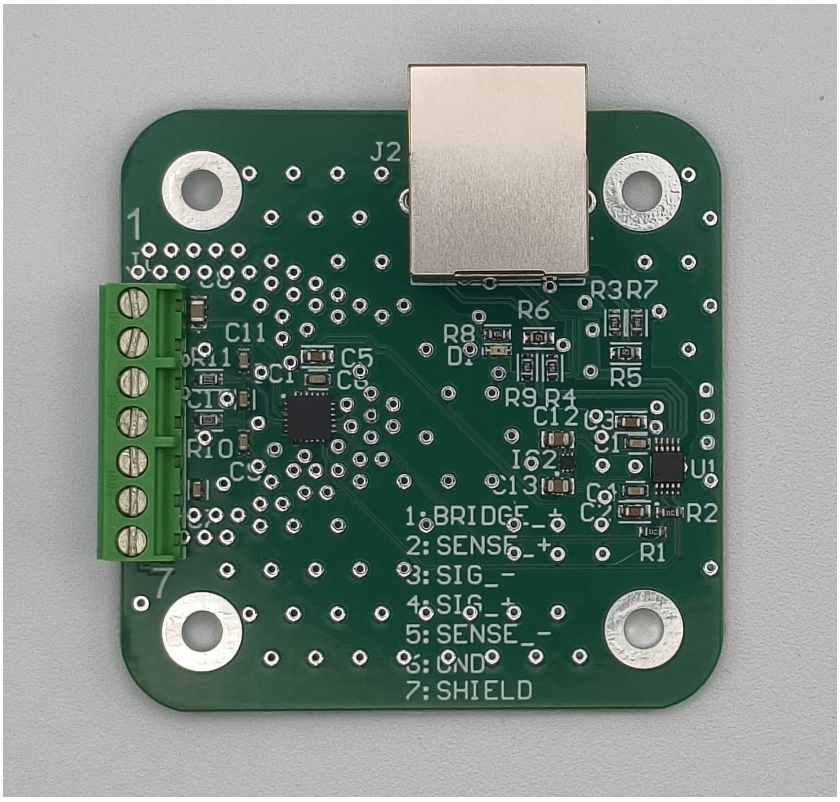


Figure 2.10: *Slave_zssc3224* assembly

Its block diagram (Figure 2.11) is similar to that of *Nuvoton NAU7802* with some added complexity.

The differential signal is generated by the sensor bridge, the load cell, external to the IC, and enters the chip through pin INP and INN. Analogously to *NAU7802*, the first element that it encounters is a multiplexer. *ZSSC3224* only offers a single input channel. The differential multiplexer is used to connect to the rest of the circuit the internal temperature sensor for temperature compensation, and to short the inputs to compensate for internal electrical offsets. The subsequent block is the differential PGA. *NAU7802* had just one stage while this IC has two amplifiers in series. The list of the possible gains of each single PGA is summarized in table 2.2. The total gain is the product of the gain of each single amplifier. The maximum achievable gain by the PGA is 216. The signal is now digitized with a 24-bit Sigma-Delta ADC. The now digitized signal is not ready yet to be output. This *Renesas* IC integrates a DSP core capable of autonomously temperature-compensating the reading and correcting for sensor nonlinearities. [13]

An internal voltage regulator creates the needed excitation voltage for the sensor bridge and the rest of the analog electronics. This voltage is fixed at around 1.6V,

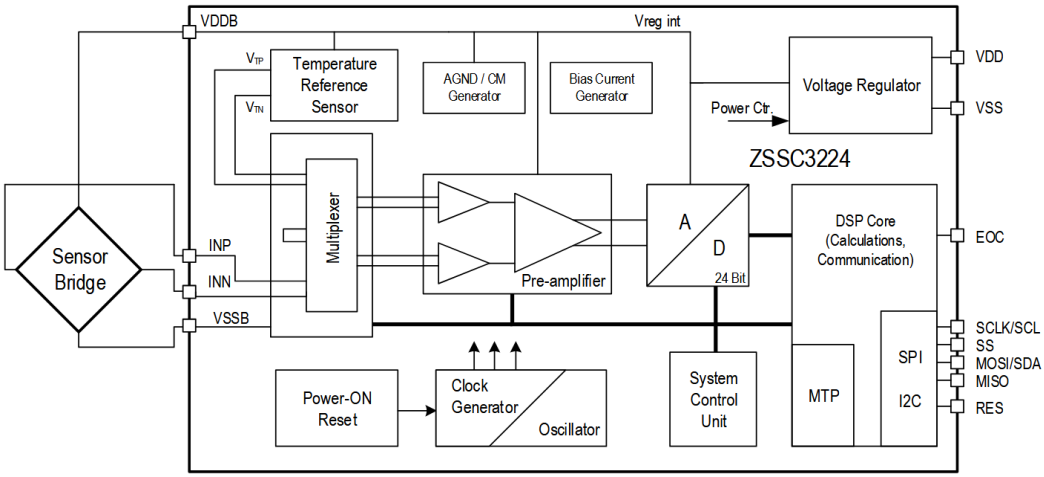


Figure 2.11: *ZSSC3224* block diagram [13]

Table 2.2: Possible PGA gain summary

Stage 1 Gain	Stage 2 Gain
6	1.1
12	1.2
20	1.3
30	1.4
40	1.5
60	1.6
80	1.7
120	1.8

only around 35% of the voltage supplied by *NAU7802*. This voltage is different from the reference voltage of the ADC that is set at 1.4V.

Front-End Communication Interface

Both *ZSSC3224* and *NAU7802* communicate through I2C. I2C is a communication protocol intended to connect multiple peripheral "slaves" to a controller "master". Only two signal lines are used: one carrying serial data (SDA) and the other carrying serial clock (SCL). Each slave device has a 7-bit address that identifies it on the bus. Each device on a single bus must have a unique address. I2C circuits

are open drain, meaning that they are only allowed to pull down the signal line. It is thus necessary to add a pull-up resistor on SDA and SCL.

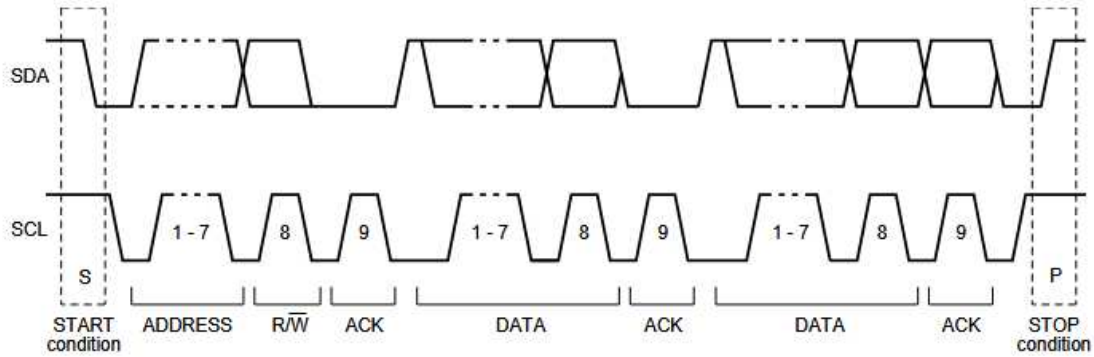


Figure 2.12: I2C signal diagram [14]

Figure 2.12 shows a generic I2C message. There are two types of frames in a message: the address frame contains the address of the target slave device, and the data frames that carry the actual data. All frames are 9 bits long, including the address frame. Since the address of each slave is just 7 bits long, the eighth bit is used to indicate if the operation type (1 is for read, while 0 is for write). The ninth bit is the acknowledgment bit. After every byte transmitted, the receiver must send an acknowledgment to the transmitter to indicate that the byte was successfully received and that the communication can continue to the next byte. In this project *Standard-mode* I2C is used, with a bit rate of 100kbit/s, but higher speeds are available, up to 5 Mbit/s with some limitations [14]. I2C is widely adopted by almost every sensor manufacturer, but it is designed just for short-range inter-IC communication on the same PCB. It lacks several essential features of purpose-built industrial communication protocols, such as robust noise immunity and extended communication range, making it unsuitable, in its normal form, for reliable communication over, possibly, meters of wire needed for the slaves to communicate with the custom aggregator unit mounted on the Raspberry Pi SBC.

NXP PCA9615 differential I2C-bus buffer with hot-swap logic solves the issue with minimal added complexity. This IC, whose block diagram is shown in Figure 2.13, converts the standard I2C signals into stronger differential signals to be reliably transmitted through twisted pairs over longer distances even in the presence of noise, while being completely transparent to the master and the slave devices.

Since SDA and SCL are now transformed into a differential pair each, there is a need for two pairs of twisted pair cable to carry those signals, plus ground and the supply voltage. A total of 6 conductors.

A Cat 5 Ethernet cable was selected. This cable, which is composed of four twisted pairs, is adequate for this project and leaves one pair of conductors to be

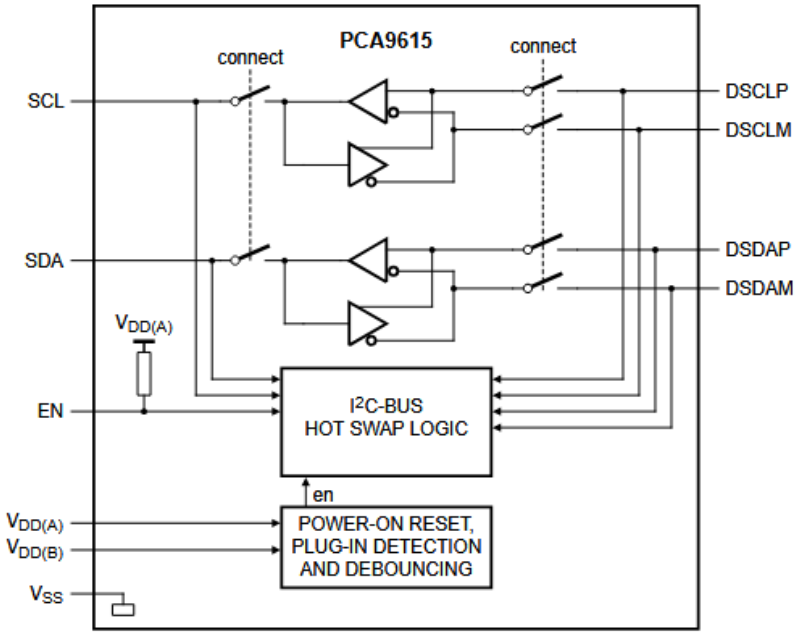


Figure 2.13: *PCA9615* block diagram [15]

used for additional functions.

Being of widespread use, its cost is lower compared to other alternatives.

Using an Ethernet cable, the simplest connector solution to implement is the RJ45 connector. They are locking connectors that ensure reliable transmission and prevent accidental disconnections.

It is easy and inexpensive to find on the market pre-fabricated assemblies using Cat 5 Ethernet, or comparable alternatives, and RJ45 connectors to be used for this project, ensuring modularity, adaptability, and ease of deployment.

2.2 Sensor Data Aggregator

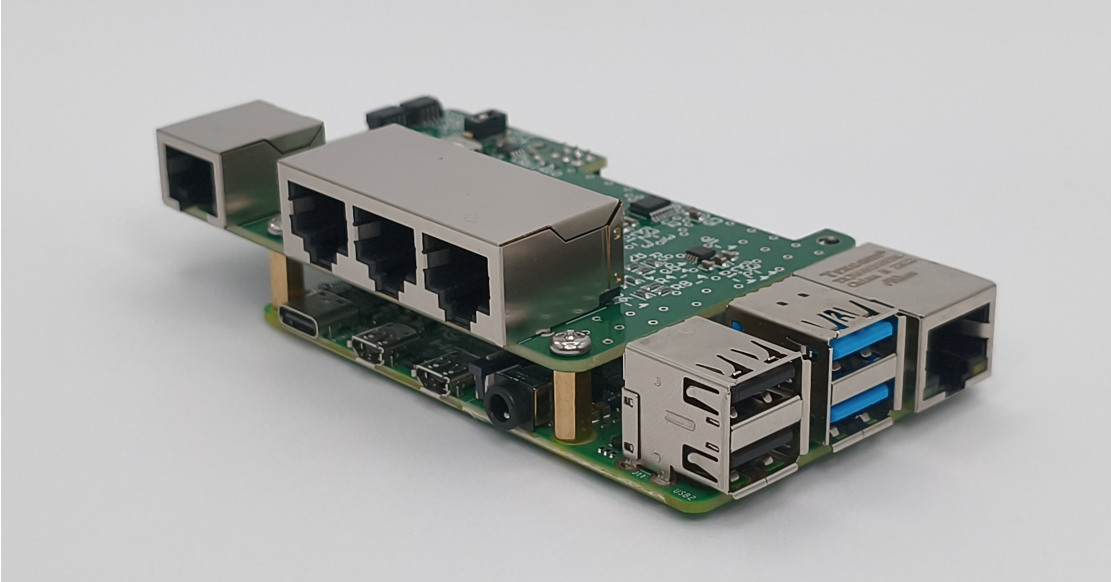


Figure 2.14: *Raspberry Pi 4* with the custom shield *Pi_master*

Every system needs to measure four scales; thus, four channels are needed in each growth chamber. The analog signal generated by each load cell has been digitized, and now it is available to be transmitted. There is a need for a system that collects the data from all the scales and sends it to the main database server to be stored, further processed, or displayed.

At the heart of this unit is a *Raspberry Pi* SBC, as of specifications provided, with a custom shield to interface all the scales called *Pi_master*.

2.2.1 Raspberry Pi 4 model B

For this project, a *Raspberry Pi 4 model B*, in the variant with 4 gigabytes of RAM, has been used.

Raspberry Pi 4 model B is a full computer running *Raspberry Pi OS*, a version of Debian Linux, the size of a credit card and costing around 60€.

The CPU is based on ARM architecture, specifically it is a *Broadcom BCM2711* quad-core Cortex-A72 at 1.8GHz of clock frequency. The operating system is installed on a micro SD card, which represents the only non-volatile storage available. Power is supplied through a USB-C connector. Connectivity is guaranteed by the 2 USB 3.0 ports, 2 USB 2.0 ports, gigabit Ethernet, WiFi, Bluetooth, 2 micro HDMI ports, 2-lane MIPI CSI camera, and MIPI DSI display port, and, most importantly,

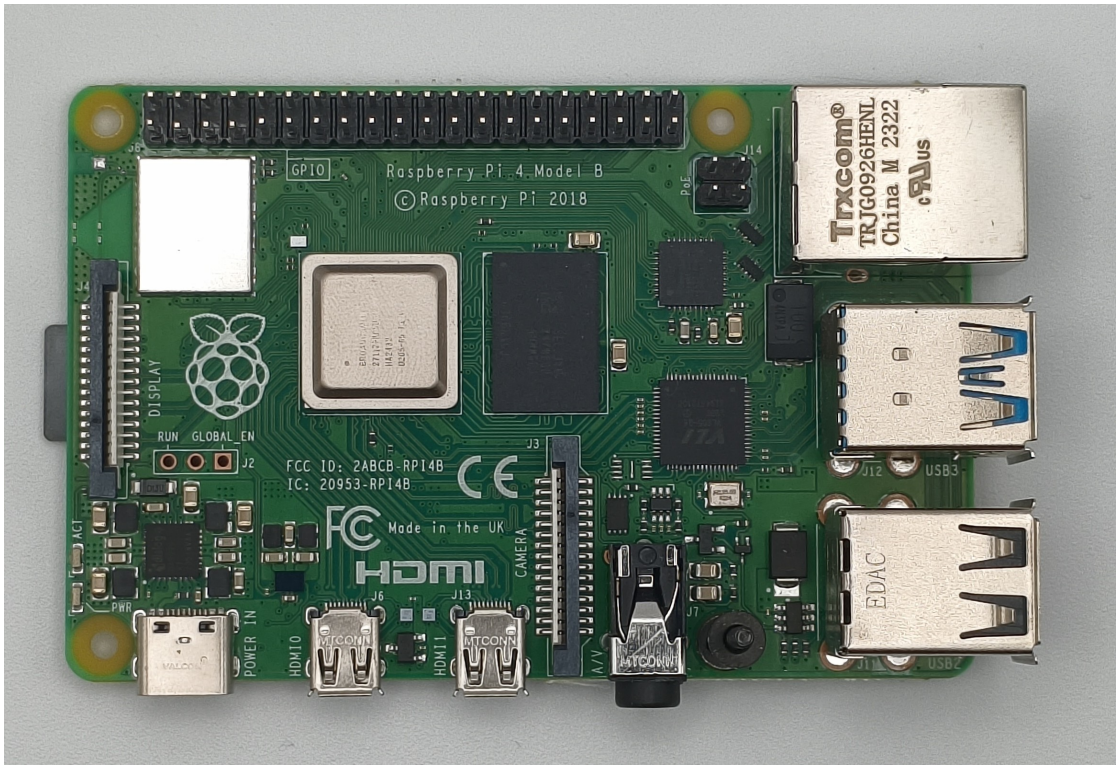


Figure 2.15: Raspberry Pi 4 model B

40-pin GPIO header. [16] This header gives direct access to an I2C bus to the CPU that can be used to interface the scales through the custom *Pi_master* shield.

A custom *Python* software runs on this SBC to interface with *Pi_master* board, read all the sensors, control actuators of various kinds, and send all the collected sensor data to the main database server.

2.2.2 Pi_Master

Pi_master is a custom shield designed to directly connect to the GPIO header on top of the *Raspberry Pi* and interface up to four scales, through RJ45, plus other external sensors or actuators using *Adafruit STEMMA QT* interface [17].

STEMMA QT is a physical standard using JST SH connectors with a 1 pitch carrying I2C signals (SCL on pin 1 and SDA on pin 2), 5V (on pin 3) and ground (on pin 4). Many sensor producers have adopted this standard interface for their sensor or have developed similar standards with the same connectors and signals, such as *Sparkfun QWIIC*, that can be cross-compatible with some little care about logic level and power supply, since *STEMMA QT*, in this implementation, operates at 5V while others operate at 3.3V. Even though Qwiic is a 3.3V standard, many devices

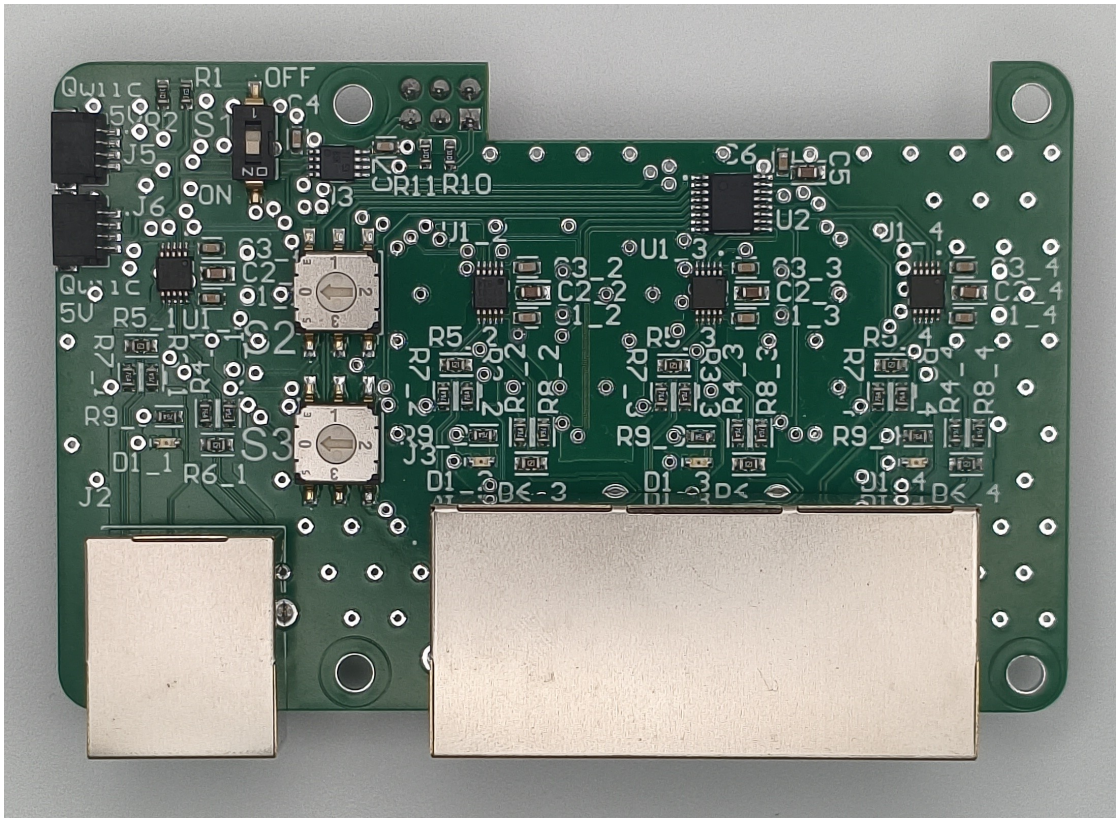


Figure 2.16: *Pi_master* assembly

that use it operate well at 5V. Additionally, level shifter boards for converting between 5V and 3.3V are readily available and inexpensive.

STEMMA QT was selected for its flexibility, for the wide availability of sensors and actuators of all kinds already using it, and for the added mechanical stability compared to the usage of standard pin headers and jumper wires.

Examples of manufacturers adopting *STEMMA QT* or *QWIIC* are Sparkfun, Adafruit, Arduino, Silicon Labs, BeagleBoard, Pimoroni, Protocentral, Seeed Studio, alongside others.

Central in *Pi_master* is the differential I2C-bus buffer IC explained in subsection 2.1.2 used to interface with the scales. In *Pi_master*, it transforms the differential I2C bus used to interface the remote scale sensors back into a standard one.

The second key IC is I2C port expander *Analog Devices MAX7320* (Figure 2.17).

This IC offers eight push-pull output pins controllable via its I2C interface [18]. A push-pull uses two MOSFETs, one N-type and one P-type, to control the output voltage by connecting the output pin to the power supply voltage or ground to

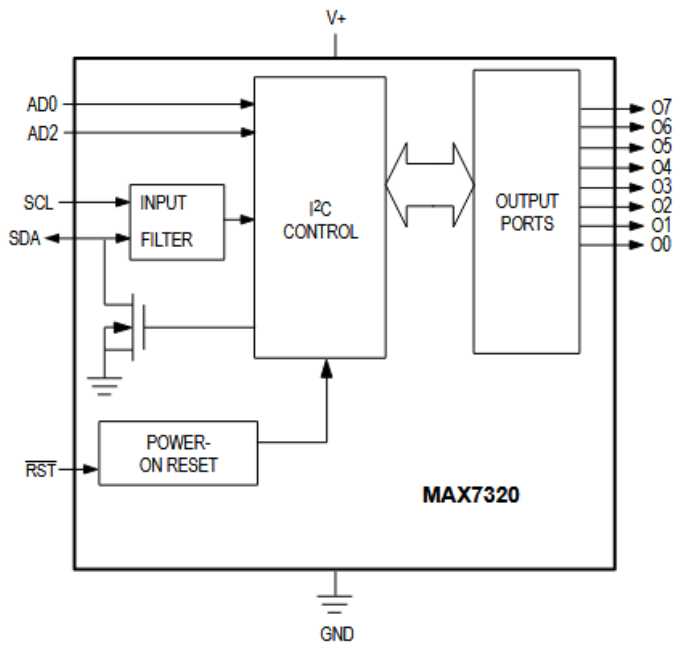


Figure 2.17: *MAX7320* block diagram [18]

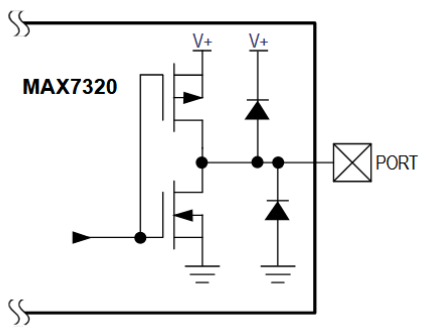


Figure 2.18: *MAX7320* output port structure

generate the two output logic levels. This IC is essential since *NAU7802* has a fixed I2C address of *0x2A* [9] that can not be changed, making it impossible to connect multiple on the same bus, while *ZSSC3224* has software define address [13] but the modification process is cumbersome and requires rebooting the IC for the change to take effect, which undermines the goal of user-friendliness of the system. *PCA9615* offers a chip enable pin that, controlled through *MAX7320*, allows for

selectively disabling or enabling single sensor connection and thus multiplexing multiple sensors with the same address on the same I2C bus.

2.3 Optional Accessories Boards

As explained in subsection 2.2.2, *Pi_master* board offers two ports for expansion. These ports have been used to connect a relay board to control water pumps used for selective and controlled irrigation of each pot under test.

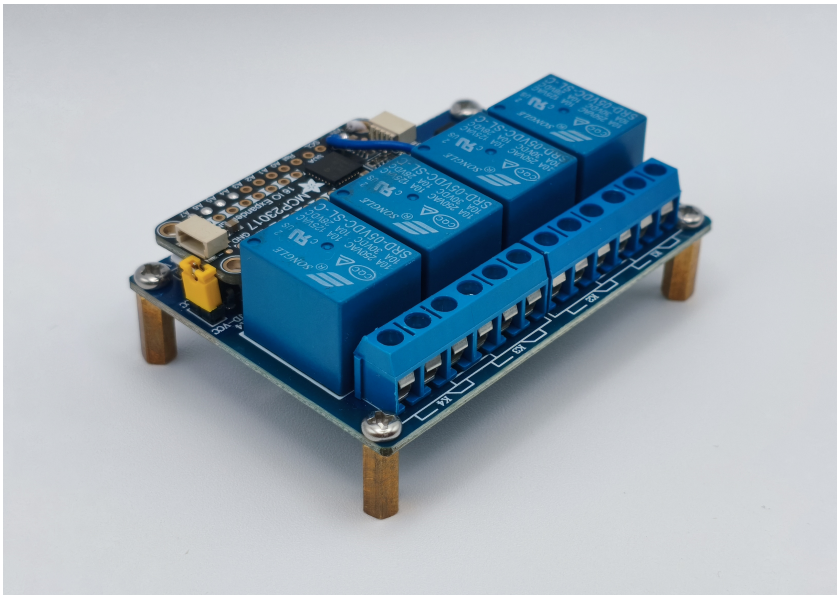


Figure 2.19: Relay expansion board

The device is based on a commercially sourced generic relay board, with a blue PCB in Figure 2.19, equipped with four relays capable of switching loads up to 10A 20VDC or 10A 250VAC. These relays are controlled by a digital signal with a 5V logic level.

On top of this board it has been soldered a module produced by *Adafruit* mounting a *Microchip MCP23017* IC using *STEMMA QT* as a communication interface. *MCP23017* is a 16-channel I2C GPIO interface (Figure 2.20) capable of working in push-pull mode when used as digital outputs.

This *Adafruit* board allows control of the relay using I2C through the *STEMMA QT* interface.

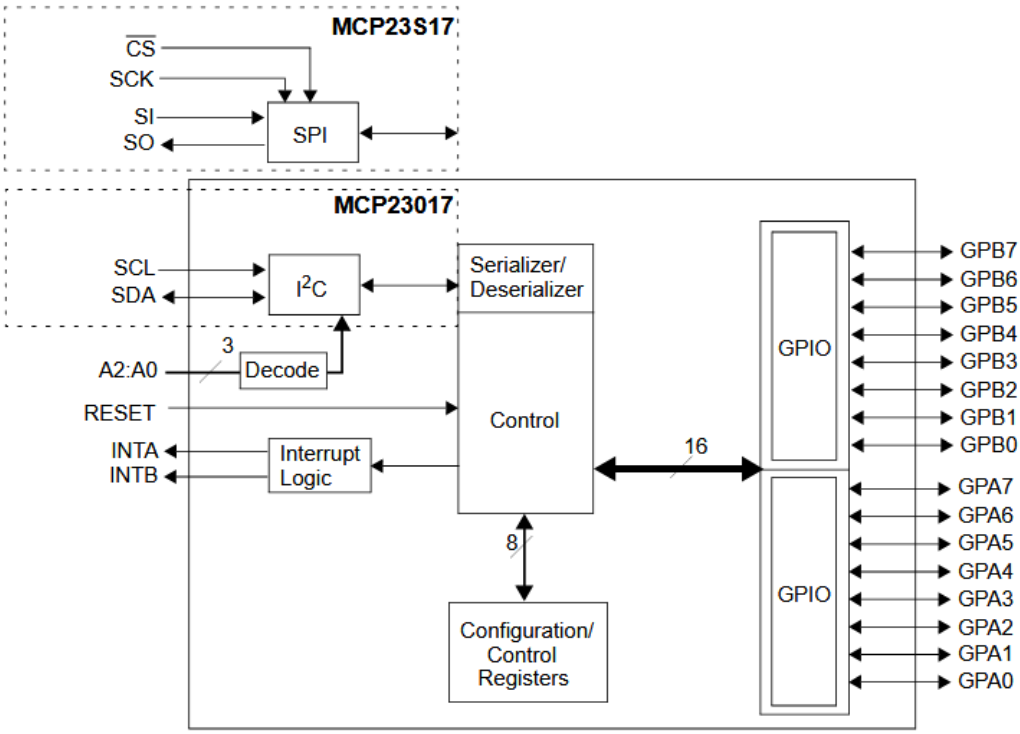


Figure 2.20: *MCP23017* block diagram [19]

2.4 Cloud Server

The digitized scales' readings are aggregated by the *Raspberry Pi* SBC on the edge and sent, together with the data from all the other scale systems installed, to a central cloud server to be managed and stored.

The transmission protocol chosen is MQTT. According to the official documentation [20]:

> "MQTT is a client-server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts, where a small code footprint is required and/or network bandwidth is at a premium."

MQTT is based on the concept of publish/subscribe, which decouples the device that produces the information, called the *publisher*, from the device that uses this

information, called the *subscriber*. All messages pass through a central server, called *broker*, that administers the communication. *Publishers* do not send messages to a specific *subscriber* but, instead, they publish them to a specific *topic*. *Topics* (for example: scales/pimaster1/device/irrigation) are hierarchical text strings that are used to identify the *subscribers* to which data is addressed; so clients that want to receive a certain data stream have to subscribe to that specific topic, and the broker will ensure they will receive the correct data.

MQTT supports three levels of QoS that controls the reliability of successful data delivery [20]:

- **QoS 0: At most once delivery.** The message is delivered depending on the conditions of the network. If the message is sent but there is no response from the receiver, the message is not resent.
- **QoS 1: At least once delivery.** The message is ensured to be delivered at least once, but duplicate receptions are allowed.
- **QoS 2: Exactly once delivery.** This is the highest level of QoS among the three. The message is ensured to be delivered exactly one time at the cost of extra resource usage compared to the other QoS levels.

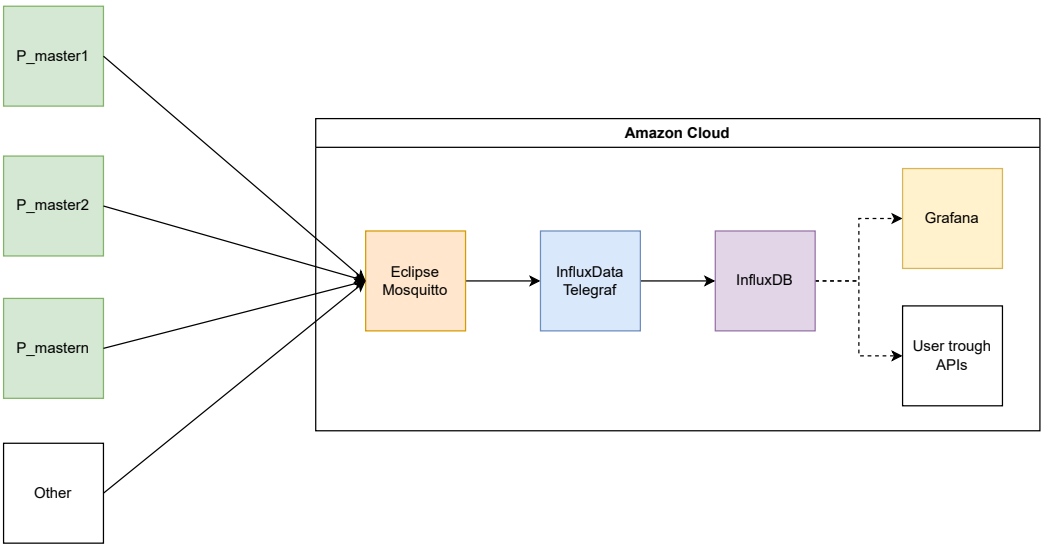


Figure 2.21: Cloud architecture block diagram

Figure 2.21 shows the basic block diagram of the system installed to manage the incoming data from all the sensor nodes.

All these services were installed on an *Amazon AWS* virtual machine.

2.4.1 Eclipse Mosquitto

Figure 2.22: *Eclipse Mosquitto* logo [21]

Eclipse Mosquitto is an open source MQTT broker characterized by being particularly lightweight on server resource usage. Being the MQTT *broker*, it administrates the MQTT network. It was chosen to self-host the broker to have full control over the user creation, user authentication management, up-time, and usage limits in the optics of future expansion of the cloud platform beyond the simple usage for the scale system.

2.4.2 InfluxData Telegraf

Telegraf is a server agent that collects data from various data sources, MQTT for this application, processes it, aggregates it, and outputs it to a compatible data-store.

The data created by the scale system, and received through MQTT by *Telegraf*, is in JSON format. JSON is a human and machine-readable text file organized in key-value pairs where each key identifies a value that can be a string, a boolean, a number, or even an array or other JSON object. *Telegraf* is capable of extracting the necessary numeric data from this JSON, and through the usage of API store it in the *InfluxDb* database.

2.4.3 InfluxDb

InfluxDb is an open-source time series database. A time series database is a particular kind of database where data points are stored and indexed in time order, making it ideal for IoT and sensor measurements. The second advantage of *InfluxDb* is its schema-less design, where the columns of the table are not defined at the design stage but are defined by the incoming data, increasing its flexibility.

2.4.4 Grafana

Grafana is an open-source platform for data monitoring and visualization. Using API keys, it is connected to the *InfluxDb* database where data is stored, and it is used to create dashboards to visualize the collected data in real time.

Chapter 3

Slave_NAU7802 Hardware Design

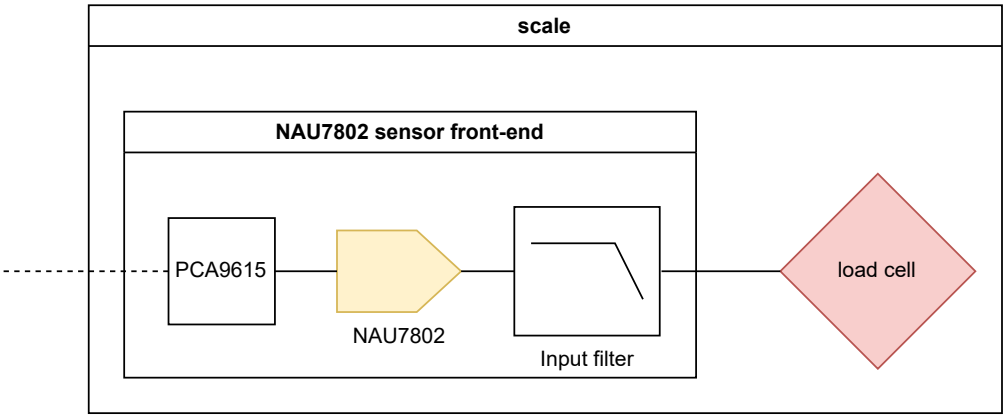


Figure 3.1: General block diagram - NAU7802 scale subsystem

The hardware analysis begins with the examination of the scale subsystem.

In section 2.1, a detailed analysis of the scale assembly chosen and the working principle of the load cell was performed. Now we will focus on the first sensor front end developed, using *NAU7802* IC.

NAU7802 sensor front-end acquisition board is built on three main functional blocks that are schematized in their simplest form in Figure 3.1. The incoming signal from the load cell first enters a filter to remove the majority of the common and single-mode noise, and *NAU7802* immediately digitizes it. The digital signal is now transmitted to Pi_master through *PAC9615*.

3.1 Schematics

The schematic analysis will follow the same fundamental blocks illustrated in the block diagram of Figure 3.1.

3.1.1 Input Filter

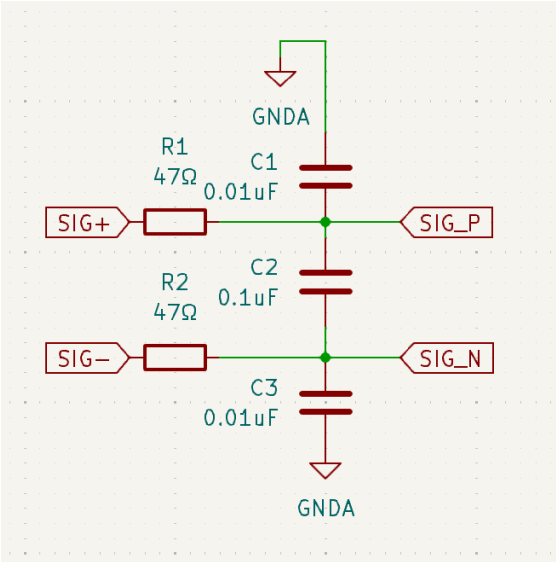


Figure 3.2: Input filter schematic diagram *NAU7802*

Figure 3.2 shows the common-mode and single-mode differential low-pass filter used to filter the signal from the scales entering *NAU7802* ADC. This filter is used to cut off all the frequencies that are greater than the maximum frequency that ADC is built to accept to avoid aliasing, and to remove some higher frequency noise from the digitized signal.

SIG+ and SIG- are the two inputs from the central node of the resistive bridge composing the load cell, while SIG_P and SIG_N are the two filtered outputs. R10 and R11 are the two input resistors that work for both the single-mode and common-mode filtering. Since the two differential channels have to behave equally when taken in isolation, these two resistors are chosen equal and indicated in the formulas as R_{IN}. Capacitor C13 is for the single-mode filtering and indicated as C_{DIF}, while capacitors C12 and C4 are for the common-mode filtering. These have, again, to be equal and they are indicated as C_{CM}.

$$f_{\mathrm{CM}} = \frac{1}{2\pi R_{\mathrm{IN}} C_{\mathrm{CM}}} \tag{3.1}$$

$$f_{\mathrm{DIF}} = \frac{1}{2\pi * \left(2R_{\mathrm{IN}}\right) * \left(C_{\mathrm{DIF}} + \frac{1}{2}C_{\mathrm{CM}}\right)} \tag{3.2}$$

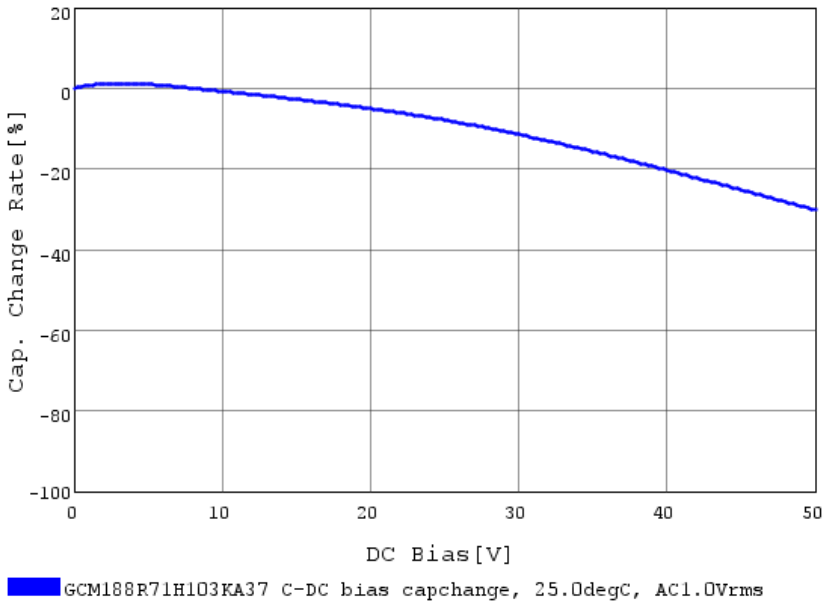
Following the indication from the book *"Analog Engineer's Pocket Reference Guide"*[22], common mode capacitors were selected around ten times greater in value compared to the differential capacitor, allowing for a differential mode cutoff frequency around twenty times lower than the common mode one. This is to prevent the common-mode noise from being converted into differential noise due to component tolerance. Equation 3.1 is used to calculate the common mode cutoff frequency, around 338.6 kHz in this implementation, and equation 3.2 is used to calculate the differential cutoff frequency, in this case around 16.1 kHz. The datasheet of *NAU7802*[9] suggests values for the resistors and the differential capacitor, while the common mode capacitors are a modification from the author to attempt to improve the quality of the digitization.

Criticalities in $C_{\mathbf{DIF}}$ Capacitor Model Choice

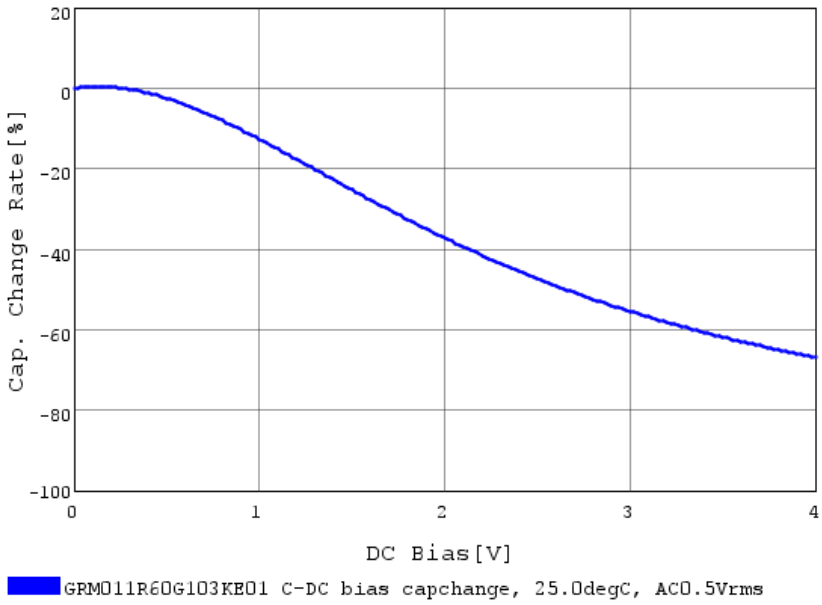
MLCC are capacitors that are well-suited for general-purpose applications for their wide availability, small dimensions, good performance, and low price. In an application like this filter, they face a significant limitation that, if not properly considered, could lower filter performance. The differential-mode capacitors are between ground and the central node of the load cell, so they have an average bias voltage across them of 2.25V.

In MLCC, the capacitance is not stable with bias voltage.

Figure 3.3 shows the relative capacitance change caused by different bias voltages of two capacitors using the manufacturer-provided data. The capacitor in Figure 3.3a is the capacitor chosen for the filter. The capacitance value of this capacitor with a DC bias of 2.25V is increased by around 1.1% [23], an irrelevant change for this application. On the other hand, Figure 3.3b shows that this second capacitor, with the same capacitance from the same manufacturer, shows a relative capacitance change of around 42.7% at 2.25V bias, making it unsuitable for the usage in a filter of this kind.



(a) Adequate MLCC *(Murata GCM188R71H103KA37J)* 0–50V x-axis scale



(b) Inadequate MLCC *(Murata GRM011R60G103KE01)* 0–4V x-axis scale

Figure 3.3: Relative capacitance change on two 0.01uF MLCC under applied DC bias voltage [23].

3.1.2 NAU7802 - ADC

As briefly explained in subsection 2.1.2 *NAU7802* is a highly integrated IC capable of performing all the operations needed to digitize the filtered load cell signal. Figure 3.4 shows the diagram of the circuit used.

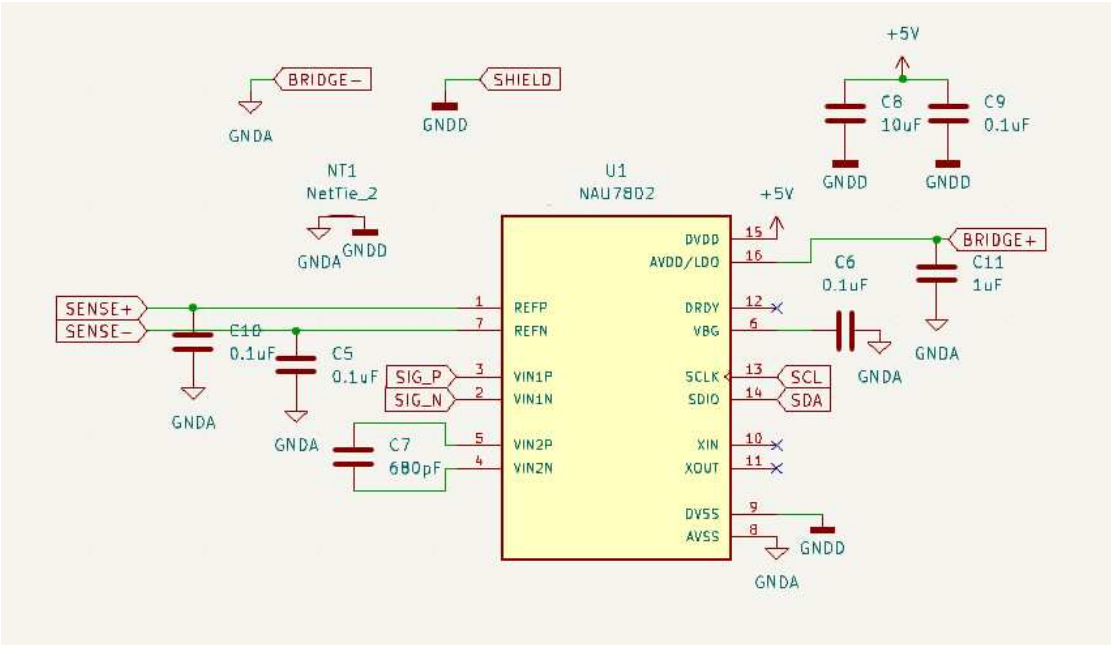


Figure 3.4: NAU7802 subsection setup for 4.5V bridge supply

The circuit has seven connections with the load cell. One is a shield to protect the delicate signals from electromagnetic interference. The six remaining are illustrated in the diagram in Figure 3.5. According to the explanation of the load cell working principle performed in subsection 2.1.1, signals BRIDGE+, SIGN_N, SIGN_P, and BRIDGE- are enough to measure this kind of resistive sensors. The addiction of SENSE+ and SENSE- might seem at first sight redundant. But it is not the case.

Knowing the excitation voltage, the capacity of the load cell, and the precise full-scale output, it is possible to perform precise weight measurement without the need for a calibration procedure using Equation 3.3.

$$W_{KG} = \left(\frac{RAW_{ADC}}{2^{ADC_BITS} - 1} \times V_{ref}\right) \times \left(\frac{W_{max}}{G \times V_{ext} \times FSO}\right) \tag{3.3}$$

W_{KG}: Measured weight (Kg), RAW_{ADC}: Raw ADC reading (-), ADC_BITS: Total ADC output bits (-), V_{ref}: ADC reference voltage (V), W_{max}: Load cell maximum rated load (Kg), G:

Input amplifier gain (-), V_{ext}: Bridge excitation voltage (V), FSO: Full scale output sensitivity (V/V)

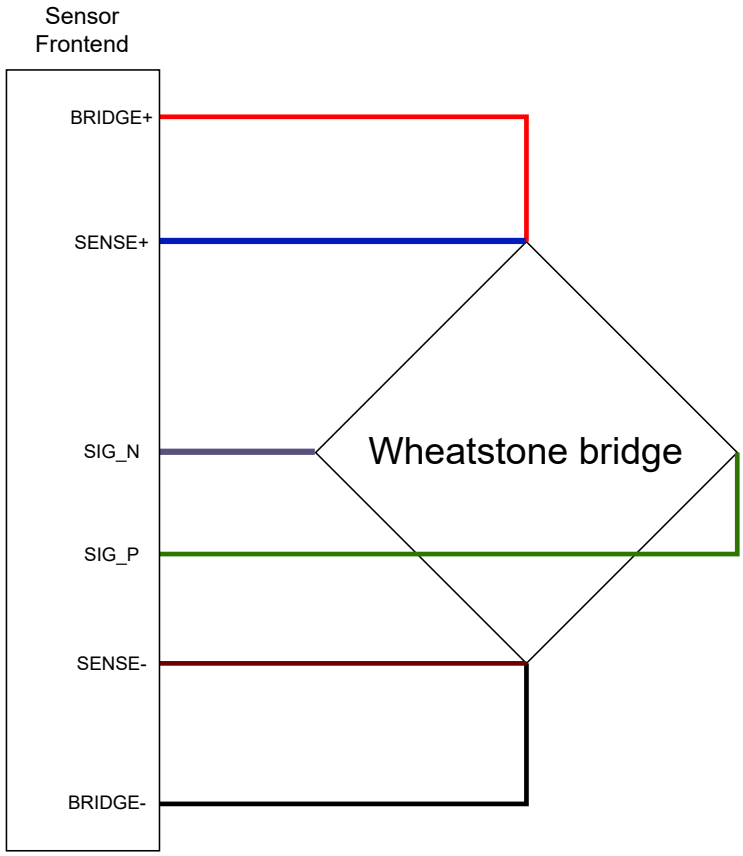


Figure 3.5: Load cell connection diagram

It is clear to see that if V_{ref} and V_{ext} are equal, they can simplify to become irrelevant in the equation. This coupling is advantageous since inaccuracies in the analog power supply do not affect the precision of the output. Load cells are devices with a relatively low input impedance, for this specific load cell, of around 400Ω [10]. This means that in BRIDGE+ and BRIDGE- lines pass some current that, because of the finite conductance of the material composing the wire itself, leads to a minute voltage drop that introduces errors in the excitation voltage at the resistive bridge. This error makes V_{ref} different from V_{ext}, making it impossible to simplify the two voltages in the calculations. SENSE+ and SENSE- are used similarly as in 4-wire resistance measurement. While BRIDGE+ and BRIDGE- are

low impedance connections that carry the current needed for the bridge to work SENSE+ and SENSE- are high impedance lines that are just used from the ADC to sense the actual voltage on this bridge, feeding high impedance input REFP and REFN of the ADC. These are the upper and lower reference voltages. The 0 output of the ADC is now not 0V anymore, but it is the actual voltage at the lower node of the bridge, and the same thing for the upper ADC reading.

Capacitors C10, C5, and C11 are used to limit the noise on the signals and the bridge excitation voltage.

BRIDGE+ is the positive excitation voltage. It is provided by a dedicated internal voltage regulator of NAU7802 IC [9], that also feeds the rest of the analog sections. This regulator can be digitally set to various voltage values according to Table 3.1

Output voltage (V)
2.4
2.7
3.0
3.3
3.6
3.9
4.2
4.5

Table 3.1: NAU7802 regulator voltage settings

Since the digital power supply of the IC is 5V, 4.5V analog power supply was selected. This is the highest setting, making the output voltage of the bridge greater, thus helping to reduce the relative effects of noise on the measurements.

Only one of the two available ADC channels is used; capacitor C7 is added to the second channel. *NAU7802* offers a setting to use this capacitor to further filter the output of the PGA, increasing noise performance and increasing the equivalent number of bits of the ADC specifically at high PGA settings. The value of such a capacitor is suggested in the datasheet [9] and here reported in table 3.2.

Analog supply Voltage (V)	**C_{filter} (pF)**
3.3	330
4.5	680

Table 3.2: Recommended C7 values for different voltages

VBG, pin 6 of the IC, is the output of the internal band gap reference. In the datasheet, it is asked to bypass it with a high-quality 0.1uF capacitor for optimum

performance. Capacitor C6 is used for this purpose.

Capacitors C8 and C9 are power supply decoupling capacitors. Decoupling capacitors are used to filter noise and spikes from the power supply lines and to provide local energy storage, stabilizing power supply lines. Power planes and cables have parasitic inductance. A sudden spike of required power, such as when a switch closes, causes the voltage of the supply to suddenly drop. Decoupling capacitors compensate for this inductance.

3.1.3 PCA9615 - Digital Data Transmission

Key in the digital data transmission chain is PCA9615. Figure 3.6 shows the circuit used to implement the, possibly, relatively long-distance digital interface over I2C.

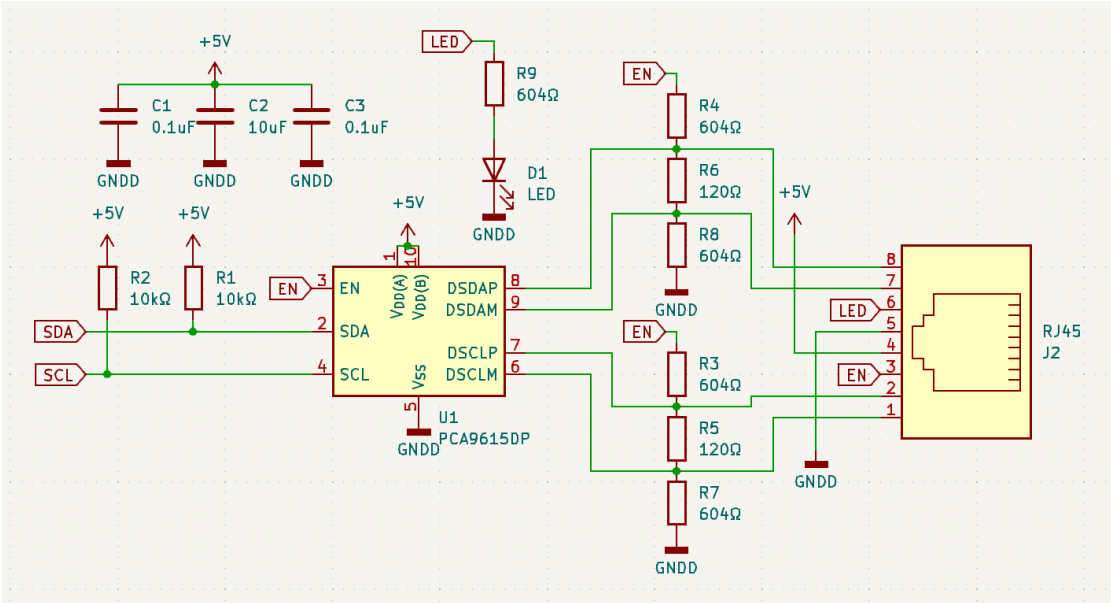


Figure 3.6: Digital interface section schematic

NAU7802 ADC is connected to the SDA and SCL lines. Two pull-up resistors, R1 and R2 are used, one on each I2C line. Pull-up resistor values have been selected considering the maximum input current in each IC and the bus capacitance following the procedure explained in *Texas Instruments SLVA689* [24]. The absolute maximum input current for the SDA and SCL lines in PCA9615 is 80mA, and for NAU7802, the only other device connected to this I2C bus, the datasheet does not directly specify it. To be safe, in this project, 3mA was considered the maximum allowable input current. The bus capacitance has three main factors contributing. The input/output capacitance of the PCA9615 is between 7 and 10 pF [15]. The input/output capacitance of NAU7802 is, again, not specified in its datasheet [9]. Comparing various I2C devices on the market, it was assumed to be around 10pF maximum. The last contribution is the capacitance of traces on the PCB. Using software *Saturn PCB Design Toolkit Version 8.41* from *Saturn PCB design INC*, the bus capacitance was calculated with the actual final PCB characteristics as around 0,84pF/cm (Figure 3.7). On the final PCB, the I2C traces connecting PCA9615 and NAU7802 are around 4cm in length, accounting for around 3.36pF of capacitance. The total bus capacitance is thus around 23.36pF, in the worst-case scenario.

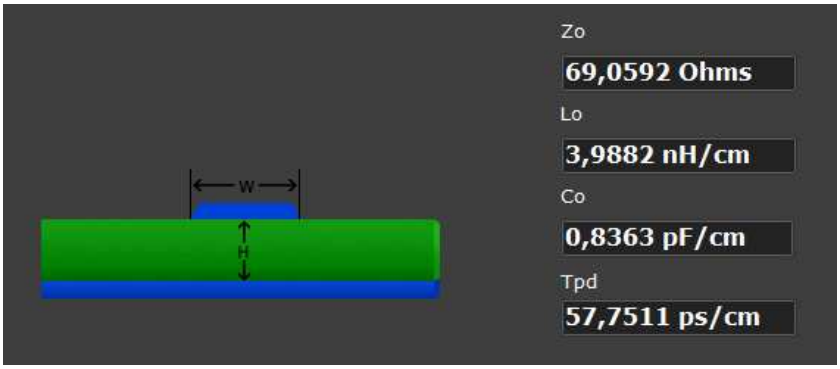


Figure 3.7: Output results of *Saturn PCB Design Toolkit Version 8.41*

Using Equation 3.4 and 3.5, it is possible to calculate the upper and lower bounds of pull-up resistance. In this specific case, the upper bound is 50.5kΩ while the lower bound is 1.5kΩ. 10kΩ is a good compromise and was selected.

$$R_p(max) = \frac{t_r}{(0.8473 \times C_b)} \tag{3.4}$$

$R_p(max)$: Maximum pull-up resistance (Ω), t_r: Maximum rise time (s) (1000ns for Standard Mode I2C), C_b: Bus capacitance (F)

$$R_p(min) = \frac{(V_{CC} - V_{OL}(max))}{I_{OL}} \tag{3.5}$$

$R_p(min)$: Minimum pull-up resistance (Ω), V_{CC}: Supply voltage (V), $V_{OL}(max)$: Maximum logic low voltage (V) (generally 0.4V)

Following the digital signal path through PCA9615, introduced in subsection 2.1.2, the single-ended SDA and SCL are transformed, respectively, into the differential DSDAP, DSDAM, and DSCLP, DSCLM.

The differential SDA and SCL are still open-drain signals, meaning that they can just be pulled down by the connected IC and require external pull-up resistors to be pulled up. The required pull-up network is formed by R4, R6, and R8 for the differential SDA signal and by R3, R5, and R7 for the differential SDL. The differential pairs found in an Ethernet wire have 100 Ω impedance; to obtain optimal performance and range, this impedance should be matched by the transmitter and receiver assemblies. The two resistor networks double as an impedance matching circuit. The datasheet of PCA9615 suggests the value of 120Ω for R6 and R5 and 600Ω for R4, R8, R3, and R7. 600Ω resistors are not common, so they were replaced by 604Ω resistors, standard in the E96 series.

The two differential signals are connected to the first and last two pins of an RJ45 socket.

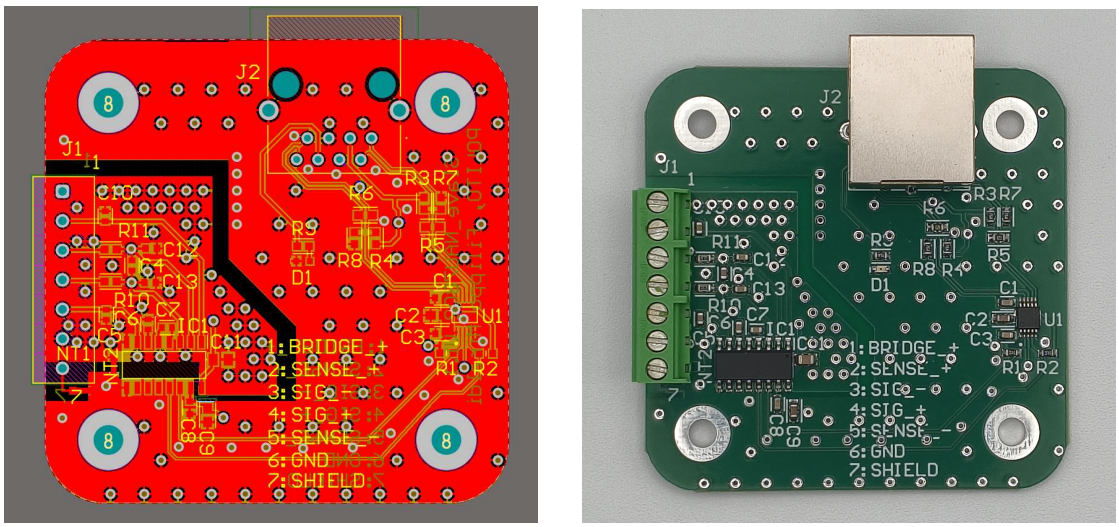
The two just-created networks should be directly connected to the digital power supply of the circuit, 5V in this case. This is not ideal since they represent a series of 3 resistors between 5V and GDNDD of equivalent value of 1328Ω. Taken in isolation, it is not a problem, but the final system has 4 channels, each composed of 2 digital signals with two sets of resistor networks (one on each end of the differential signal line). This is equivalent to sixteen 1328Ω resistors between the power supply and ground in parallel. The equivalent resistance of this set of resistors is 83Ω. It does not serve any function except when data is digitally transmitted, wasting power for the rest of the time. Since this is a mains powered device, this is not a big problem, but if it is possible, it is better not to waste power.

As already briefly explained in subsection 2.2.2, there is a need for an enable signal to turn on and off a specific section of the circuit in the *sensor data aggregator*. Since the Ethernet cable is composed of four pairs of wires and only three are strictly needed, one of the two remaining cables can be used to propagate this enable signal also to the slave and turn on and off the pull-up network and PCA9615 IC. This signal is indicated by EN.

The second unused wire is connected to a LED, making it software controllable. The EN signal is 5V in amplitude. R9 was selected of 604Ω to reduce the number of different components used.

C1, C2, and C3 are decoupling capacitors C2 is a 10uF used as bulk decoupling, while C1 and C3 are 0.1uF and used one for each of the two power supply pins of the PCA9615.

3.2 PCB Implementation



(a) Nau7802 PCB project file **(b)** Nau7802 final PCB implementation

Figure 3.8: Side by side view of the project file and the final PCB manufactured

3.2.1 Electrical Design

Figure 3.8 shows the project file and the manufactured PCB side by side. On the left of the PCB, the green screw terminal is used to connect the load cell, while the top metal square connector is the RJ45 port used to transmit the digital readings.

The board shape is square with rounded corners, size 55mmx55mm.

It has been used a four-layer PCB, meaning that it is formed of four independent copper layers separated by fiberglass material. Each copper layer can be independently etched to create the required circuit geometry. Vias are used to connect one layer to the other.

The PCB stackup selected is *JLCPCB JLC04161H-7628* (Figure 3.9) since, due to the lack of any other special requirement, it was the most cost-effective controlled impedance four-layer stackup available. The total thickness is 1.6mm.

The two outer layers are 1oz copper, meaning that the copper layer is 0.035mm thick, while the two inner ones are 1/2oz, meaning they are just 0.0152mm thick. The fiberglass separating the first and last two layers, prepreg, is almost 1/5 the thickness of the central core of fiberglass.

For this specific PCB, it was chosen to use the two inner layers as ground planes, while the two outer layers for signal and power.

2) JLC04161H-7628 Stackup

Layer	Material Type	Thickness	
Top Layer	Copper	0.035mm	
Prepreg	7628*1	0.21040mm	
Inner Layer L2	Copper	0.0152mm	1.1mm H/HOZ with copper
Core>	Core	1.065mm	
Inner Layer L3	Copper	0.0152mm	
Prepreg	7628*1	0.21040mm	
Bottom Layer	Copper	0.035mm	

Figure 3.9: *JLCPCB JLC04161H-7628* stackup

Every time a signal passes through a trace on the PCB, an equal and opposite return current is generated. At high frequency, this current will pass on the ground plane immediately beneath the signal trace as skin effect dominates, while at low frequency, it will tend to take the shortest route, always on the ground plane. Since the signal plane is different from the inner ground plane, where the return current passes, the two conductors form a loop. According to Faraday's law, a time-varying current through a loop will radiate an electromagnetic field proportional to the area of the loop. The same effect is present on two-layer PCBs using a separate signal layer and ground layer, but on a four-layer PCB signal and ground are separated by just 0.21mm of prepeg, while on a two-layer PCB it's 1.6mm, 7.6 times more, leading to a much greater loop area and thus more radiated emissions. Removing the ground layer completely will only worsen the situation since the return path will be coplanar with the signal and no more on an uninterrupted plane, thus not controlled as in the previous cases, causing greater loop areas and emissions.

The plane where the return current of a particular signal passes is called the reference plane.

Every time a signal has to move from the top to the bottom layer, a via is used. A via is a plated hole that connects to planes. Particular care has been taken in putting at least one ground via next to each signal via. This ground via is used to change the reference layer of the return current to the most adjacent to the signal. Without this via the current will have to change the layer anyway, and it will do it through the closest ground via, if none are available through some IC or component, leading to interference and an increase in radiated emissions.

Electromagnetic emissions have to be minimized.

The European Union, under its Harmonized Standards framework, enforces a specific directive, Directive 2014/30/EU, regulating the electromagnetic compatibility of devices [25] but, as stated in the *Blue Guide on the Implementation of EU Product Rules 2022*, "Union harmonisation legislation applies to products which

are intended to be placed (and/or put into service) on the market" [26]. Since this scale system is not meant to be placed in the market, but it is just a prototype to be used exclusively in the lab, it is possible to pay less attention to the legal aspects.

The emission problem can not, anyway, be overstated. Emissions still have to be minimized as much as possible since they are a problem from a signal integrity standpoint, too. These radiated emissions can couple into the sensitive analog electronics and add unwanted noise.

The presence of the two inner ground layers of the four-layer PCB, combined with the ground pour on the top and bottom layers, helps create an inner shielding effect.

The board has been populated with an array of tens of ground vias, called stitching vias, to avoid the adjacent ground plane from having slightly different voltages that might create oscillations and induce noise in the circuit.

To further limit the possibility of noise from one part of the circuit propagating to the more sensitive analog parts, the digital section has been placed on the far right, far away from the analog section on the left.

It is very clear from picture 3.8a that the uninterrupted ground plane on the inner two layers and the ground pour on the top layer is, actually, interrupted all around the analog section. In this specific PCB, the split ground approach is used. A dedicated analog and digital ground are present. These two grounds connect only at one point, under the ADC, the SOP-16 IC on the bottom left corner. The split ground approach allows for finer control of the return current and isolates the noisy return current generated by the digital section from the delicate circuitry of the analog section.

As explained in subsection 3.1.3, the differential communication protocol uses 100Ω impedance-controlled lines; this means that the PCB traces also have to be impedance-controlled at 100Ω. Using a four-layer PCB is advantageous since the top layer and the inner layers' ground plane are very close together, limiting the required width of the traces. The differential impedance line width and spacing were calculated using the built-in tool of *Altium 2025* and manually fine-tuned to obtain trace characteristics compatible with the pin spacing of the *TSSOP10* package used by PCA9615, and limiting the manufacturing cost. The final trace width is 0.223mm, and the spacing is 0.219mm.

The decoupling capacitors have been placed as close as possible to the power supply pin of each IC in an attempt to limit the parasitic inductance of the connection. For PCA9615 a small copper pour has been created around the two power supply pins.

All the components, except the Ethernet port and the screw terminal to connect the load cell, use SMD technology. In SMD components are placed and soldered on the surface of the PCB. SMD components are smaller, allow for higher component

densities, allow faster and simpler automatic assembly procedures, and are, generally, less expensive when compared with traditional through-hole technology, where each component had short leads that passed through holes in the PCB and were soldered from the bottom side. The Ethernet port and screw terminal were selected using the older through-hole technology because it guarantees a greater mechanical resistance.

All passive components are 603 size, meaning that they are 1.6 x 0.8 mm in size, except for capacitor C11, which is in 805 size, meaning 2.0 x 1.2 mm in size. 603 size was selected since it is the smallest size that can be comfortably re-worked by hand, if needed, and, additionally, there are no extra charges for the required added precision of the PCB for the higher tolerances of the smaller dimensions.

3.2.2 Mechanical Design

The shape of the PCB is dictated by the mechanical mounting constraints. Underneath the scale frame are four unused holes conveniently placed in a square pattern with a 4cm side (Figure 3.10).

Figure 3.10: Scale hole square pattern

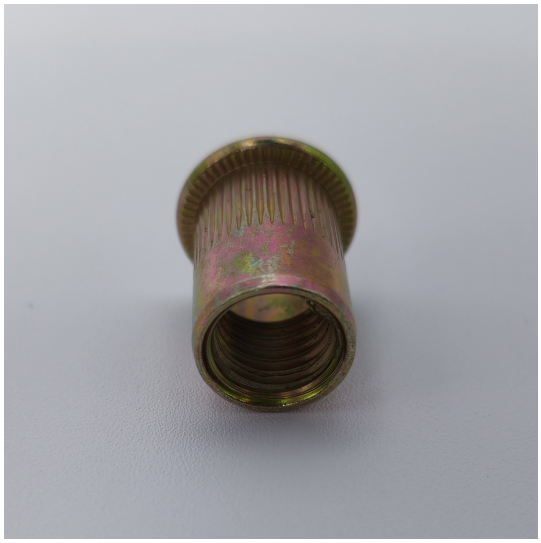
These holes can be used to mount the acquisition board as shown in Figure 3.11, giving convenient access to the Ethernet port from behind the scale without interfering with the measurement process.

These holes are blind and 5mm in diameter. It is not possible to use a combination of a screw and a nut to mount the PCB board, but at the same time, the wall of the metal profile used to create the scale is not thick enough to cut resistant threads that would, anyway, be too big since the screw will be M5 or bigger. The ideal solution to mount the board is to use rivet nuts. Rivet nuts are a type of fastener that is essentially a rivet with a threaded hole in the center. The rivet nut is inserted in the hole, and using a specially designed instrument (Figure 3.12c), the non-threaded portion of the rivet nut is deformed (the serrated section in

Figure 3.11: Board mounted underneath the scale

Figure 3.12a), causing it to expand and grip the material securely. The smallest rivet nuts available in the market have an external diameter of 5mm and accept an M3 screw, standard size for electronic standoffs. A short standoff can, then, be directly screwed on the rivet nut. Mounting the PCB in this way under the scale keeps it raised from the surface and away from possible water spills and humidity.

(a) Rivet nut side view



(b) Rivet nut bottom view

(c) Special instrument to mount rivet nut

Figure 3.12: Views of the rivet nut and mounting tool used in the project

Chapter 4

Slave_ZSSC3224 Hardware Design

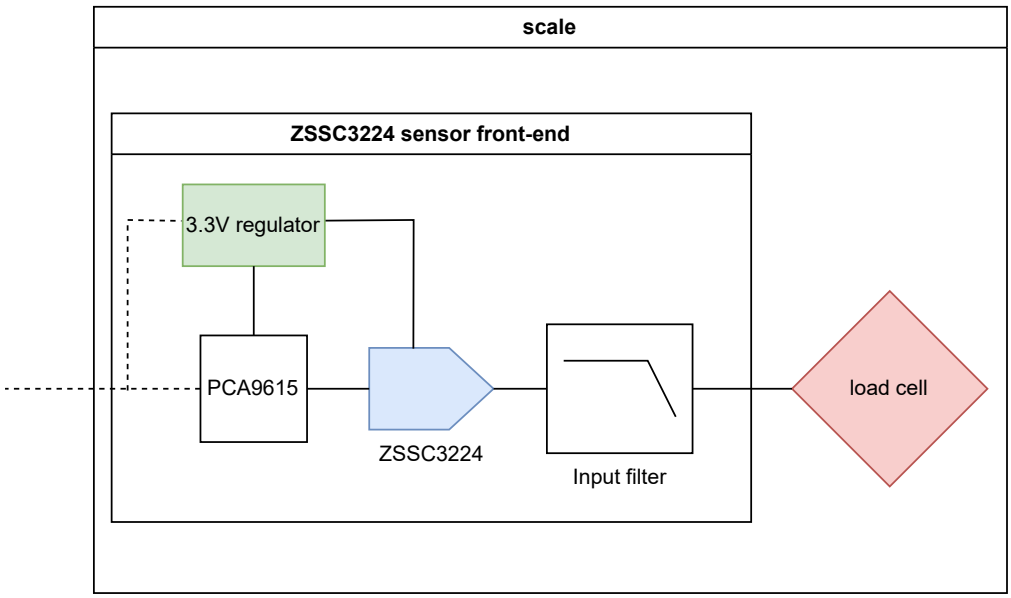


Figure 4.1: General block diagram - ZSSC3224 scale subsystem

Slave_ZSSC3224 was created to have an alternative to Slave_NAU7802. The sensor front end is the single element with the greatest impact on overall system performance. Having two different alternatives allows us to select the most performing one among the two.

The overall system architecture of Slave_ZSSC3224 is similar to the architecture

of Slave_NAU7802, explained in chapter 3, and differs mainly for the presence of a 3.3V regulator since Renesas ZSSC3225 operates at 3.3V instead of the 5V used by the rest of the system.

The signal enters the ZSSC3224 sensor front end from the load cell, and it is immediately filtered, digitized, and transmitted using the PCA9615 IC.

4.1 Schematics

Following the block diagram in Figure 4.1, the electronic schematic is divided into four different blocks.

4.1.1 Input Filter

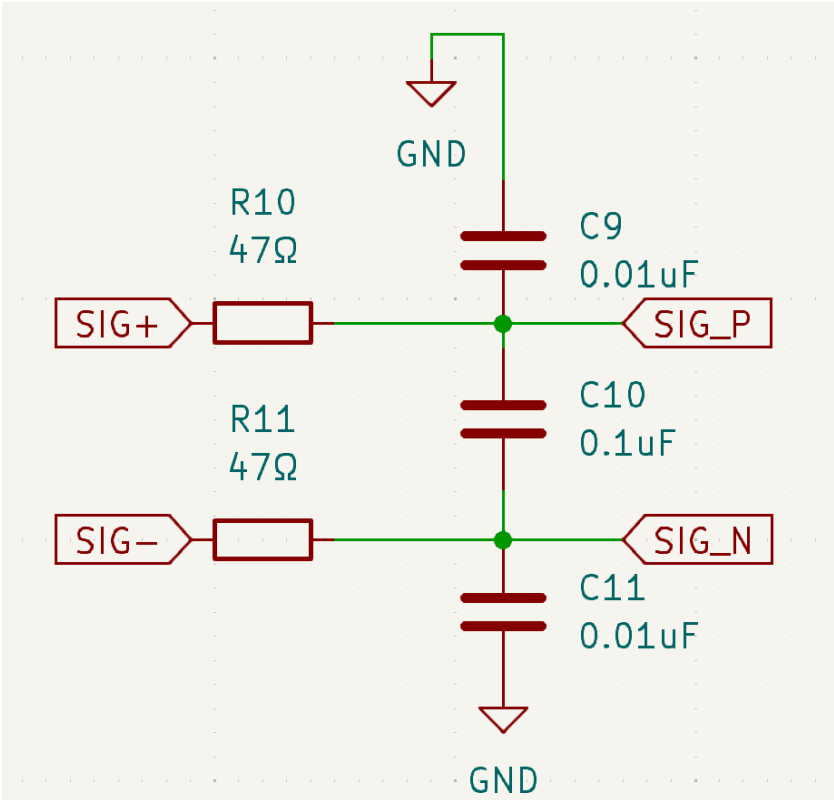


Figure 4.2: Input filter schematic diagram ZSSC3224

Since ZSSC3224 operates at frequencies close to those used by NAU7802, to reduce part count and assembly costs, it was chosen to use the same design input filter (Figure 4.2) used in Slave_NAU7802. Please refer to 3.1.1 for a more detailed analysis.

4.1.2 ZSSC3224 - ADC

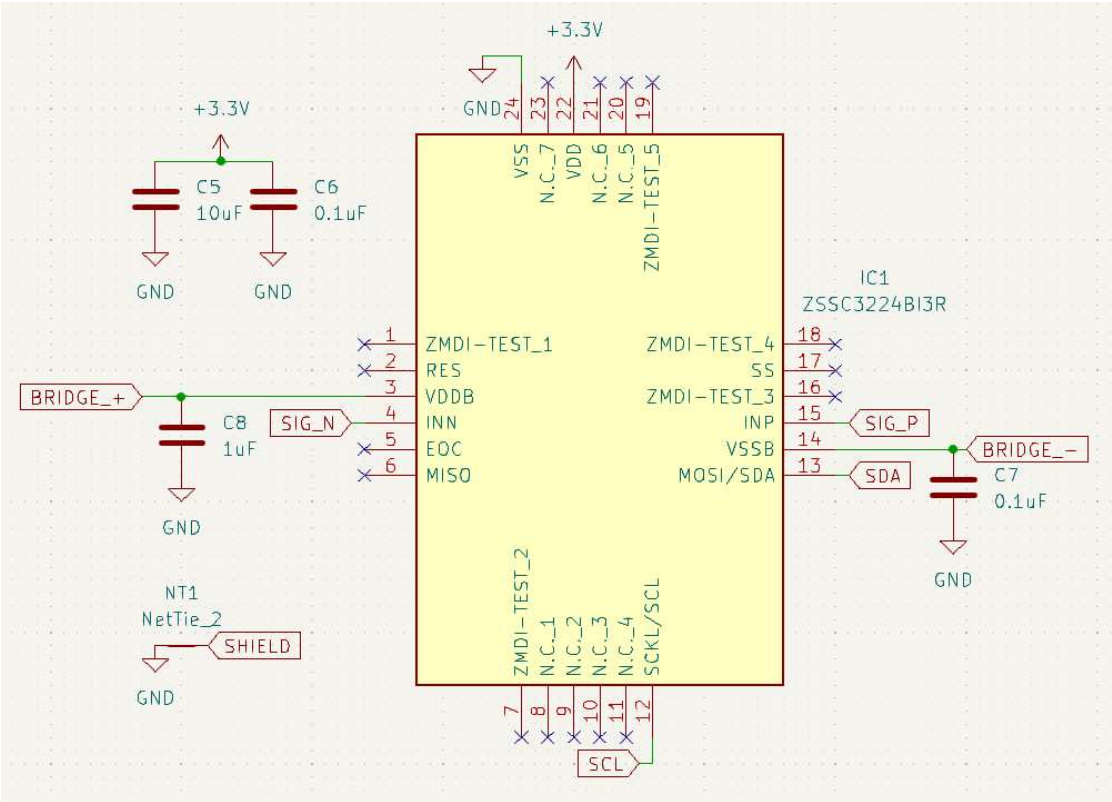


Figure 4.3: ZSSC3224 subsection circuit diagram

ZSSC3224 works differently compared to NAU7802. While in NAU7802 the positive bridge excitation voltage was provided by the integrated LDO while the negative excitation voltage was ground, or better, the analog ground in this specific implementation. ZSSC3224 has two separate pins for the two excitation voltages (BRIDGE_+ and BRIDGE_-). This is done because the analog ground is internal to the IC [13]. Capacitors C8 and C7 were mounted on the two bridge excitations in an attempt to reduce noise.

ZSSC3224 lacks the external positive and negative reference voltages, so only four wires are required to perform a measurement. The impact is limited by the lower excitation voltage of just 1.68 V [13] compared to the 4.5 V of NAU7802. Since the load cell still provides 7 wires, the two reference wires were tied together at the excitation voltage.

Capacitors C5 and C6 are the two decoupling capacitors for the IC.

4.1.3 3.3V Regulator

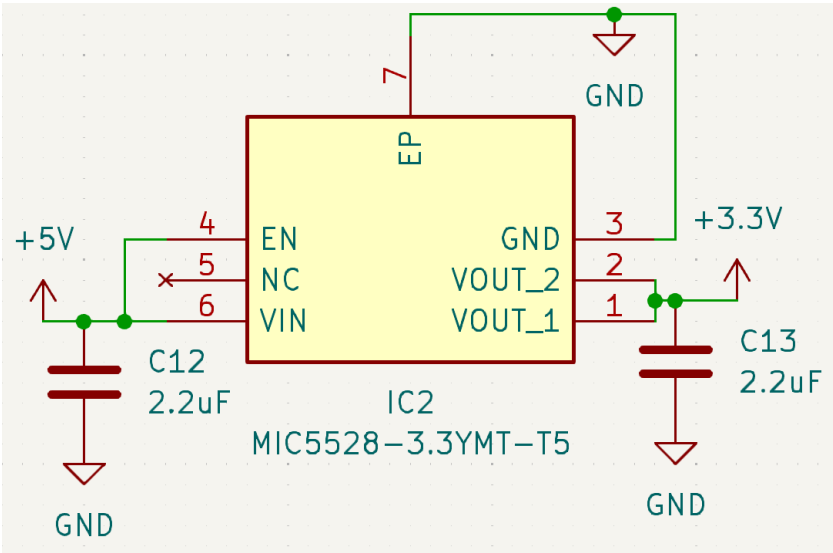


Figure 4.4: 3.3V Regulator subsection circuit diagram

Through the RJ45 connector, only a 5 V rail is supplied, but the ZSSC3224 only works with 3.3 V supply voltage.

Microchip MIC5528 is a linear voltage regulator capable of dropping the input voltage from 5 V to a stable 3.3 V. A linear regulator adjusts its resistance to maintain a steady set output; because of this, the excess energy gets dissipated through heat. It is essential to verify that the regulator does not overheat.

The actual power dissipated is expressed by Equation 4.1

$$P_D = (V_{IN} - V_{OUT}) \times I_{OUT} + V_{IN} \times I_{GND} \tag{4.1}$$

P_D: Dissipated power (W), V_{IN}: Input voltage (V), V_{OUT}: Output voltage (V), I_{OUT}: Output current (A), I_{GND}: Ground current (A)

The input voltage is 5V, the output voltage is 3.3V, and the ground current for a CMOS device is around 100uA[27]. The output current has to be calculated. According to the datasheet, ZSSC3224 consumes a maximum of 1.5 mA. The load cell has an input impedance of 406Ω with an excitation voltage of 1.68V [13]. Using Ohm's law, it is possible to calculate that 4.1 mA is used. PCA9615 supply current on pin $V_{DD(A)}$ is 16uA, while the current used by the two pull-up resistors when the I2C bus is pulled low is 0.66mA (more details in section 4.1.4). The total current is the sum of each contribution, so 6.276 mA; lower than the 500mA that MIC5528 is capable of supplying.

Filling Equation 4.1 with the just listed values, the total dissipated power on the voltage regulator is 11.1 mW.

$$P_{D(MAX)} = \frac{T_{J(MAX)} - T_A}{\theta_{JA}} \tag{4.2}$$

$P_{D(MAX)}$: Maximum dissipated power (W), $T_{J(MAX)}$: Maximum junction temperature of the die (°C), T_A: Ambient temperature (°C), θ_{JA}: Thermal resistance junction to air (°C/W)

Equation 4.2 allows us to calculate the maximum power that can be dissipated in a given ambient temperature. Since the load cell is capable of self-compensation just in the temperature range between -10°C and +40°C [10], the maximum operating temperature of the system is 40°C. Since the board is not in a box and is directly exposed to the air, it is not necessary to make any further considerations about ambient temperature. The maximum junction temperature of the die is 125°C [27] and the thermal resistance of the junction to the air is 173°C/W [27].

Filling Equation 4.2 with the just listed values, it is possible to calculate that the maximum dissipated power by the package of this regulator at 40°C is 491mW; way more than the 11.1mW that is actually dissipated. Rearranging Equation 4.2, it is possible to obtain that with the 11.1mW dissipated the body of the regulator only heats up of 1.92°C, value that in the real world application will be even lower because in the previous calculation was not considered the heat-sink effect of the heat-sink pad (pin 7) connected to the ground plane.

Pin 4 is an enable pin that allows the regulator to be turned off. Since it is not needed, it is permanently disabled by connecting it to the input voltage.

Capacitors C12 and C13, whose value is suggested by the datasheet[27], are necessary for the stability of the regulator. Ceramic capacitors capable of maintaining most of their capacity with the given bias voltage were carefully selected.

4.1.4 PCA9615 - Digital Data Transmission

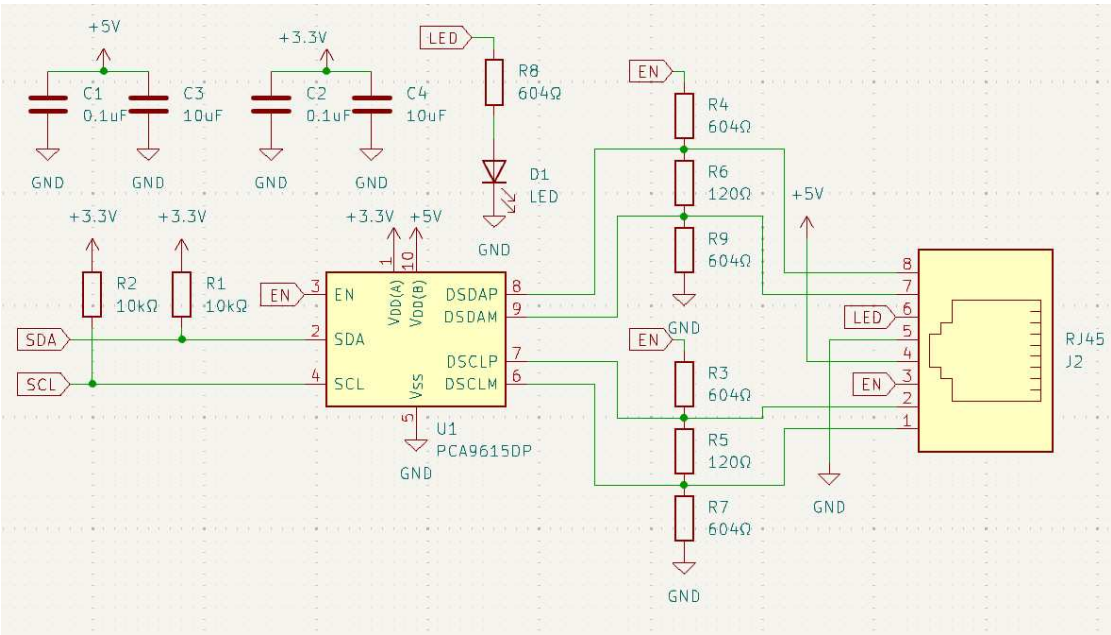


Figure 4.5: Digital interface and logic lever shifter section schematic

The circuit shown in Figure 4.5 is essentially the same circuit shown in Figure 3.6 in subsection 3.1.3 except for one minor detail that allows the 3.3V ZSSC3224 to work in a 5V system.

PCA9615 accepts different supply voltages for the single-ended ($V_{DD(A)}$) and differential ($V_{DD(B)}$) side [15]. While the LDO MIC5528 lowers the power supply voltage, PCA9615 lowers the logic level of the digital signal from 5V to 3.3V, acceptable for ZSSC3224, acting as a logic level shifter.

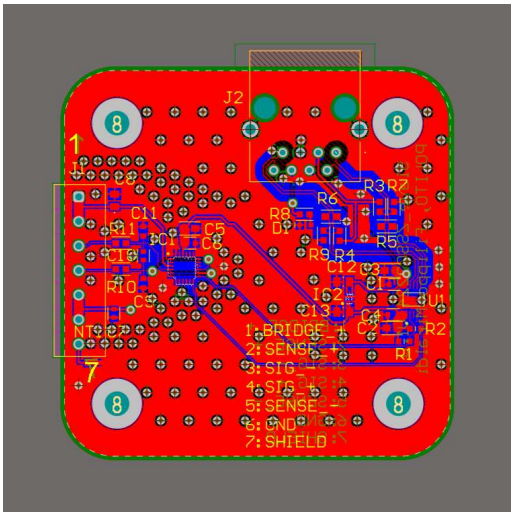
Capacitors C1, C3, and C2, C4 are the decoupling capacitors for, respectively, the 5V and 3.3V rails.

The pull-up resistors, R1 and R2, allowed range was calculated using equation 3.4 and 3.5. The bus capacitance is a maximum of 10 pF from PCA9615[15], ZSSC3224 does not specify it in the datasheet, and it is assumed of 10pF. The trace length is around 5cm, introducing 4.18pF. The total bus capacitance is thus around 25.18pF. Please refer to subsection 3.1.3 for a more detailed explanation.

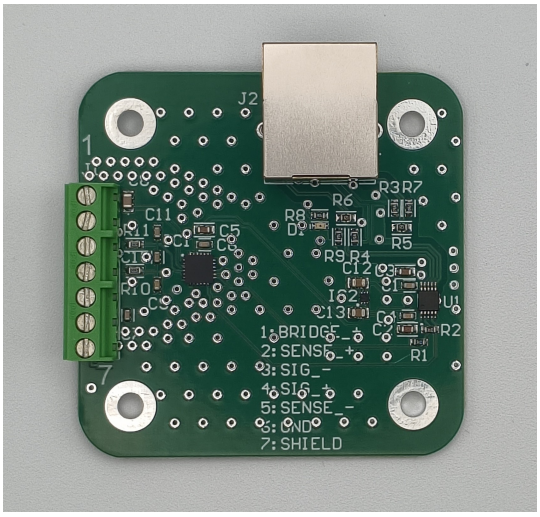
The maximum pull-up resistor value is 46.87kΩ while the minimum is 966Ω. 10KΩ was selected again to reduce component count.

4.2 PCB Implementation

4.2.1 Electrical Design



(a) ZSSC3224 PCB project file



(b) ZSSC3224 final PCB implementation

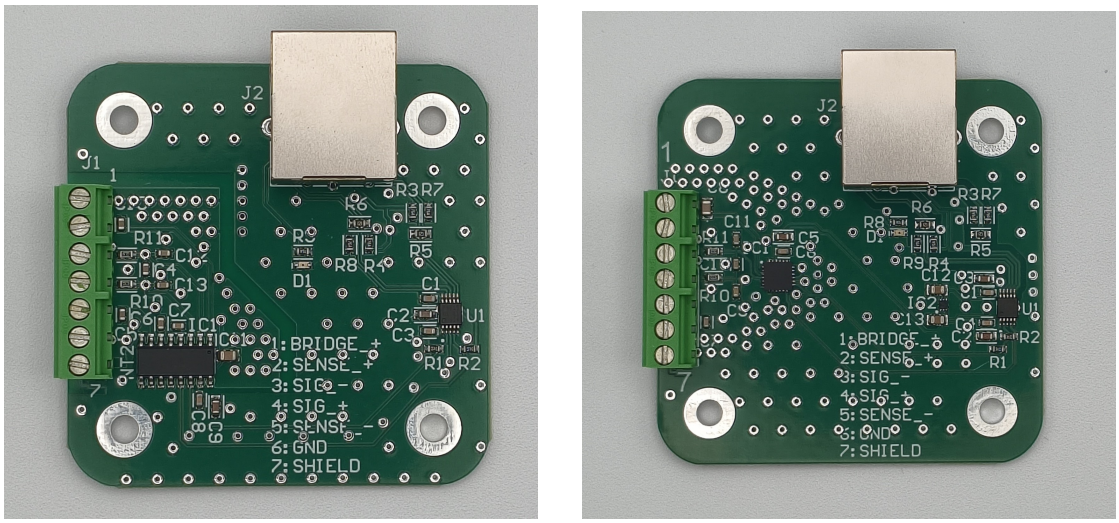
Figure 4.6: Side by side view of the project file and the final PCB manufactured

The layout of Slave_ZSSC3224 is very similar to the layout of Slave_NAU7802. Both share the same general component placement. The added voltage regulator has been placed next to PCA9615 in the digital section (bottom right corner).

It is missing the digital and analog ground, replaced by a single ground on both the analog and digital sections of the circuit. ZSSC3224 does not have separate pins for the analog and digital ground. This increases the chances of a signal crossing passing over the two different ground planes, causing more problems than simply avoiding the plane separation. Particular care has been, anyway, put into trying to optimize the return current path and avoid placing sensitive analog electronics in those areas.

Almost all passive components are 603 size with some 805 exceptions.

4.2.2 Mechanical Design



(a) NAU7802 final PCB implementation **(b)** ZSSC3224 final PCB implementation

Figure 4.7: Side-by-side view of the two analog sensor front ends

As clearly visible from Figure 4.6, Slave_ZSSC3224 shares the same physical dimensions and connector placement as Slave_NAU7802. This has been done to make the two designs immediately interchangeable with one another.

Chapter 5

Pi_Master Hardware Design

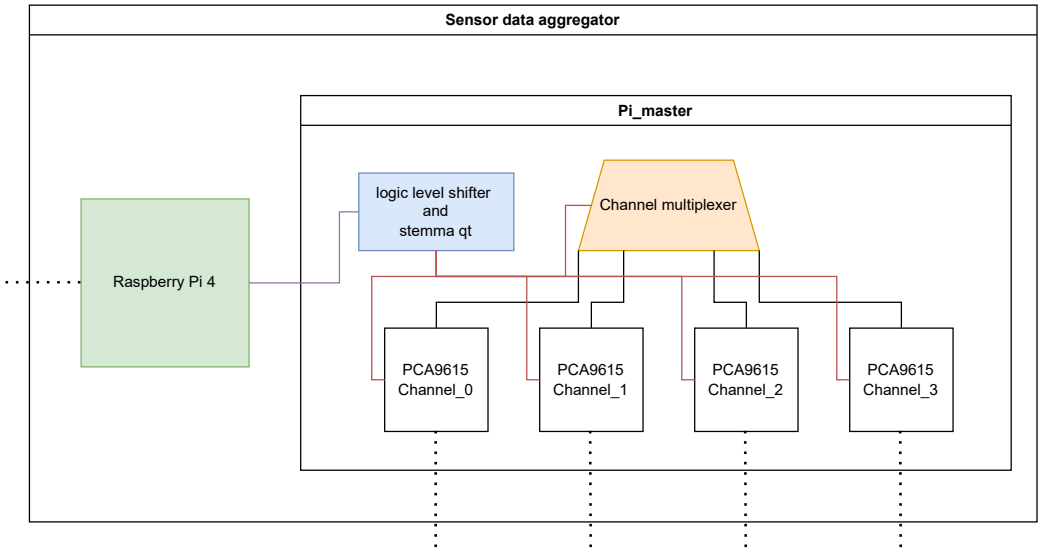


Figure 5.1: General block diagram - sensor data aggregator subsystem

Pi_master is conceived as a shield to be directly mounted on a Raspberry Pi 4 model B SBC. Raspberry Pi 4 was part of the specifications of the project, but it is an ideal computer for this application since the open source philosophy of the producer gives full access to all sorts of technical documentation, both on the software and hardware of the device. It can directly interface with various digital communication standards such as I2C thanks to the integrated GPIO. The power consumption, between 2.5W and 7W depending on the load, is low compared to a standard computer; an ideal characteristic for a computer used for long-term monitoring.

Before talking in detail about how the Raspberry Pi is used, it is necessary to take a step backward and go back to where the digital signals from the load cells were left and analyze the interface circuits built in the custom board Pi_Master as shown in its most essential blocks in Figure 5.1.

5.1 Schematics

5.1.1 PCA9615 - Digital Data Transmission

The digital signal out of the digital data transmission module of the sensor front ends travels through the possibly long Ethernet cables. It reaches one of the four identical channels identified in Figure 5.1 as PCA9615 Channel_0, PCA9615 Channel_1, PCA9615 Channel_2, PCA9615 Channel_3. These channels are identical in construction and connected in parallel to the same I2C bus.

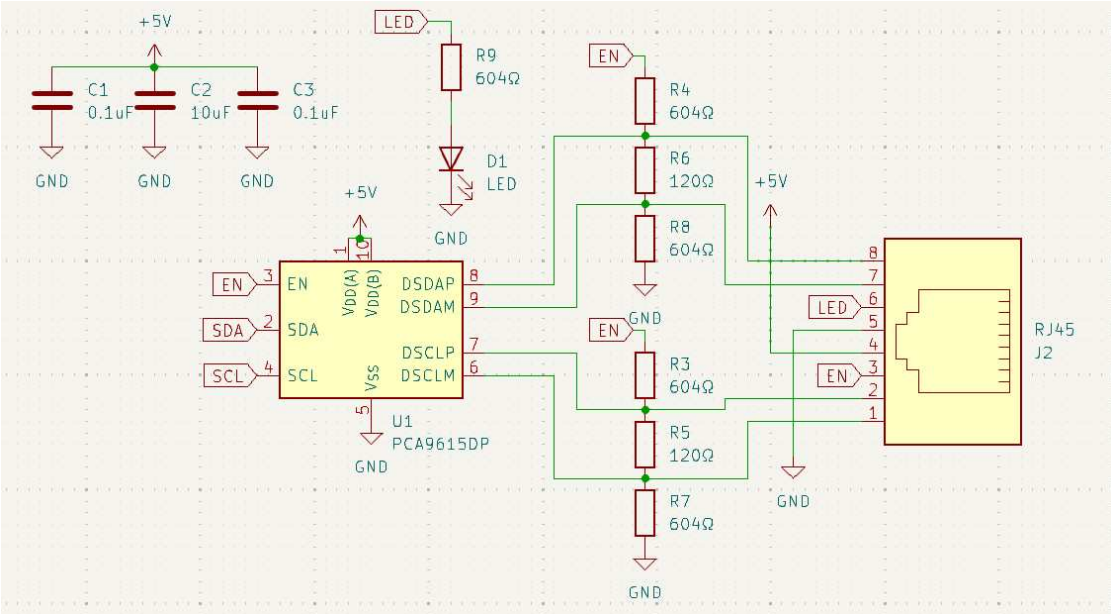


Figure 5.2: One channel schematic of the digital interface section

Schematic in Figure 5.2 shows the schematic diagram of each single digital interface channel of Pi_master. This schematic is identical to the digital interface section of Slave_NAU7802 with the two pull-up resistors removed (Figure 3.6). The reason they are identical is that they perform the same task, transforming the I2C bus, bidirectional by nature, into a differential I2C bus without logic level shifting. The two pull-up resistors are not required since four channels are connected in parallel to the same I2C bus, and a single pair of pull-ups is generally enough for a bus of this kind.

Please refer to subsection 3.1.3 for a more detailed analysis.

It was decided to add al LED, D1, also on the Pi_master to help identify both sides of a wire connecting a scale. The LED on the Slave and Pi_master side are connected to the same control signal.

5.1.2 Channel Multiplexer

On an I2C bus, all the devices connected to the same bus are required to have a unique address. As briefly explained in subsection 2.2.2 NAU7802 IC has a static I2C address of 0x2A while the address of ZSSC3224 IC can be changed, but it is a complex process that can not be feasibly repeated every time a scale is connected to the Pi_master board or moved to a different port.

Since it is not possible to change the addresses, it is necessary to multiplex sections of the bus, disabling the unused ones and only activating the needed ones at the moment of the reading. PCA9615 IC offers an enable pin that disables the differential channel, thus effectively cutting off the attached sensor board from the I2C bus. I2C is not natively designed to support hot-swap and might lead to problems. PCA9615 integrates hot-swap logic to prevent problems [15].

Since the cables connecting the various scales have two unused conductors, this enable signals is propagated in one of these wires to all the *Slave* boards, allowing to power off the differential pull-up of the transmission channel and the PCA9615 also on the slave side, just as a precautionary measure.

This enable signal could be generated by the integrated GPIO of the Raspberry Pi, but it was chosen not to do so to reserve the possibility to chain multiple Pi_Masters on the same SBC if more scales need to be connected. Having the enable signals connected to fixed GPIO pins would make the chaining of various modules impossible. The solution is to use a I2C port expander providing I2C controllable push-pull outputs.

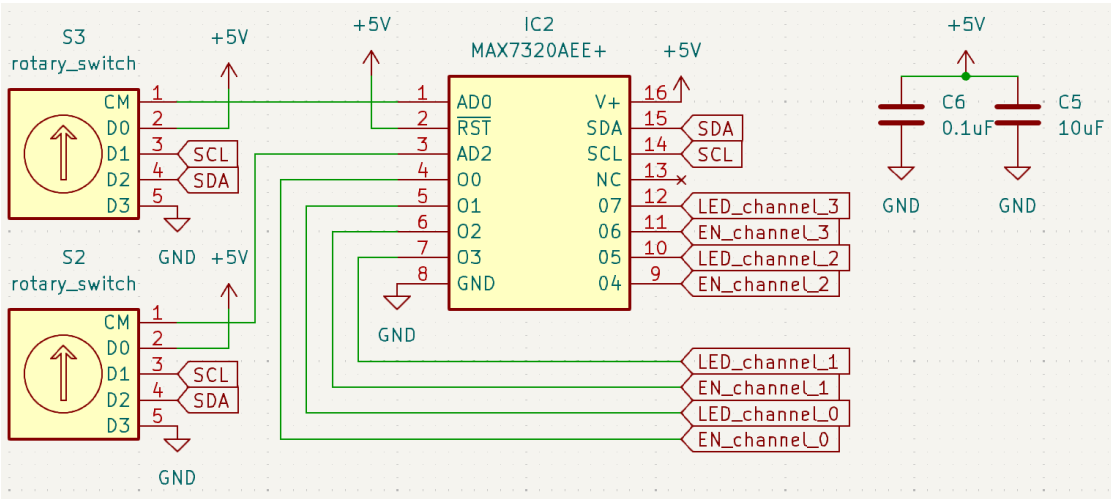


Figure 5.3: Channel multiplexer circuit diagram

Figure 5.3 shows the circuit used to implement the channel multiplexing. *Texas Instruments MAX7320* was selected to be the I2C port expander. It provides eight

symmetric push-pull outputs controllable via I2C. This IC was selected even though it is an eight-controllable port device instead of just four, as strictly needed, because it was impossible to find a smaller IC with the needed electrical characteristics and with selectable addresses. The cost difference was also negligible.

The four extra ports have been used to control the LED present on the digital data transmission module, both on the Pi_master and Slave side. This LED is used, in this implementation, to indicate the channel in use, but in future slave boards, this signal can be used for a secondary enable or even, since MAX7320 can output up to 25mA on each port, to selectively power small sections of the circuit.

The outputs of the port expander must be symmetric since it was selected, what essentially is, a high-side switching for the pull-up network and the LEDs. Given the higher cost of the high-side switch, the majority of I2C GPIO and port expanders do not offer push-pull outputs or offer quasi-push-pull, where the push-up switch is significantly weaker compared to the pull-down switch.

There are two reasons why this choice has been made:

1. **PCA9615 pin EN internal pullup:** PCA9615 features an internal pullup resistor between pin EN and VCC, 5V in this case. The IC would enable for a logic 1 and disable for a logic 0. The transmission line pullup network, if connected to low-side switching, would be disabled for a logic 1 (since the bottom of the resistor network shown Figure 3.6, Figure 4.5 and Figure 5.2 would connected to 5V as the top side node) and enable with a logic 0. The behavior of the pull-up network and the IC is thus opposite, requiring a separate line for each of the two signals controlled with different logic, increasing the chances of human error in the firmware.

2. **Flexibility:** Instead of driving the pull-up network and the IC separately, it is possible to have an extra controllable line to be adopted in future implementation of the circuit or to simply identify the scales with the LED without the need to enable it. In a possible future scenario, high-side switching ensures that power does not get backed into unwanted parts of the circuits while keeping just one ground on the Slave side.

Rotary switches S2 and S3 are used to select the I2C address of MAX7320 according to Table 5.1. Address switching is needed in case multiple Pi_master boards need to be connected in series or if there is an overlap with the channel multiplexer address and the address of an optional accessories board. The two rotary switches can connect AD0 and AD2 to ground, 5V, SDA, or SCL, giving MAX7320 sixteen possible selectable addresses.

Table 5.1: MAX7320 Address Map

S2 VALUE	**S3 VALUE**	**DEVICE ADDRESS**						
		A6	A5	A4	A3	A2	A1	A0
1	3	1	0	1	0	0	0	0
1	0	1	0	1	0	0	0	1
1	1	1	0	1	0	0	1	0
1	2	1	0	1	0	0	1	1
2	3	1	0	1	0	1	0	0
2	0	1	0	1	0	1	0	1
2	1	1	0	1	0	1	1	0
2	2	1	0	1	0	1	1	1
3	3	1	0	1	1	0	0	0
3	0	1	0	1	1	0	0	1
3	1	1	0	1	1	0	1	0
3	2	1	0	1	1	0	1	1
0	3	1	0	1	1	1	0	0
0	0	1	0	1	1	1	0	1
0	1	1	0	1	1	1	1	0
0	2	1	0	1	1	1	1	1

RST pin is the reset. It is an active low pin. Since it is not necessary to reset this IC except at boot, it was decided to hardwire this pin to the power supply to prevent accidental reset.

Capacitors C6 and C5 are the decoupling capacitors for the IC.

5.1.3 Logic Level Shifter and Stemma QT

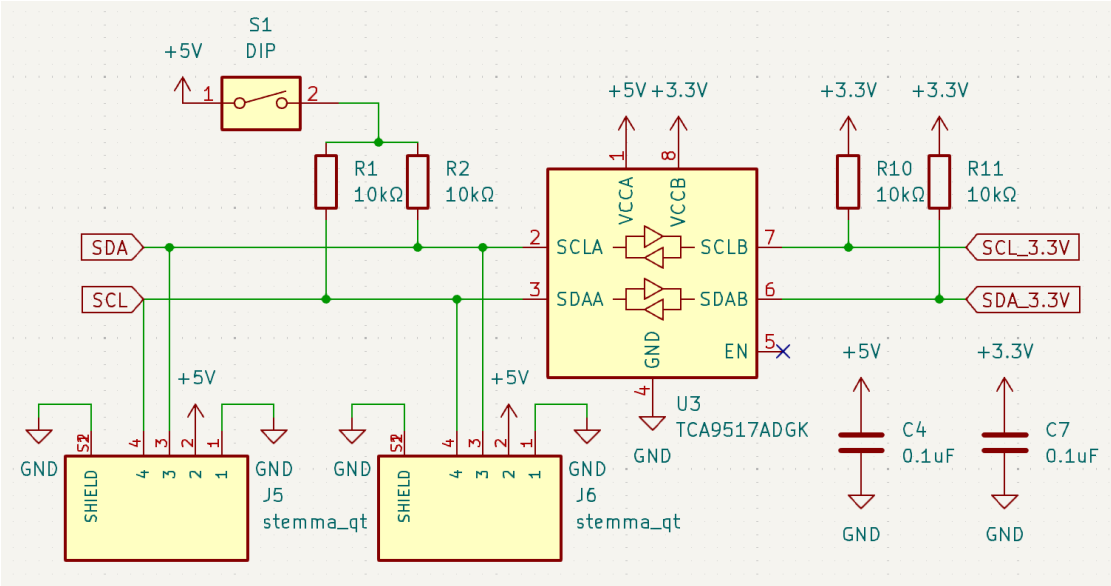


Figure 5.4: Logic level shifter and STEMMA QT ports circuit diagram

Continuing following the signals from the scales, in section 5.1.1, it left the PCA9615, where it was transformed back into a single-ended, standard, I2C signal at a 5V logic level. Now it enters the last block before the SBC.

The circuit diagram is shown in Figure 5.4.

The first two elements present are connector J5 and J6. These are the two *SM04B-SRSS-TB(LF)(SN)* used by the *Stemma QT* standard. More details are provided in subsection. 2.2.2. These connectors carry power supply, ground, and the two I2C signals representing the two main expansion ports for Pi_master. Various sensors or other Pi_master boards can be connected to these ports and share the same control SBC. The only rule is that no of the expansion boards must have the same address as the devices connected to the RJ45 ports since the expansion port is not multiplexed and will lead to conflicts on the bus.

To calculate the pull-up resistor, it was considered that there are just 6 ICs connected to the bus: four PCA9615, one MAX7320, and one TCA9517 that will be explained soon. PCA9615 contributes to a bus capacitance of 10pF maximum, being four they provide 40pF, MAX7320 contributes with 10pF and TCA9517 around 13pF. On the PCB there are around 10cm of I2C traces accounting for around 8.36pF. The total bus capacitance is 71.36pF. The acceptable pull-up resistance range for the SDA and SCL lines is 12.5kΩ and 1.5kΩ. 10kΩ were again selected.

A dip-switch has been added to eventually disable the pull-up resistor on the

line. Many QWIIC or Stemma QT boards feature soldered-on pull-up resistors. Multiple expansion boards can be connected in series, leading to their pull-up resistors being effectively connected in parallel, decreasing the equivalent resistance and increasing the current that has to pass through the devices during the I2C communication. The dip-switch can be opened to reduce the equivalent parallel resistance.

Pi_master operates at a logic level of 5V, but the Raspberry Pi 4 GPIO operates at a logic level of just 3.3V, and it is not compatible with the 5V logic level of the rest of the board.

It was decided to use a 5V logic level because the datasheet of PCA9615 [15] states date the differential communication section operates best at 5V. It was possible to adopt the same solution used in Slave_ZSSC3224 where it was used the level shifting functionality of PCA9615 but it was preferred not to because it meant powering the expansion ports with 3.3V, since the entire single-ended I2C bus would have been at a 3.3V logic level. This would have made it impossible to concatenate multiple Pi_Master using the expansion ports since there would be no possibility for the 5V required to pass from one board to the other.

The second deciding factor is that many sensors available on boards equipped with Stemma QT or Qwiic accept 5V input, and in the remaining cases, it is quite easy to move from a higher voltage to a lower voltage using a small and inexpensive linear regulator and a logic level shifter. Commercial solutions are available and inexpensive. On the other hand, moving from a lower voltage is more complex. There are commercial boards available that use a charge pump and a level shifter, but charge pumps have severe limitations. They are electrically noisy and limited in the output current. Adafruit produces both the logic step-up and step-down Stemma QT converter. The two cost the same, but the step down can manage 500mA of current, while the step up can manage just 100mA.

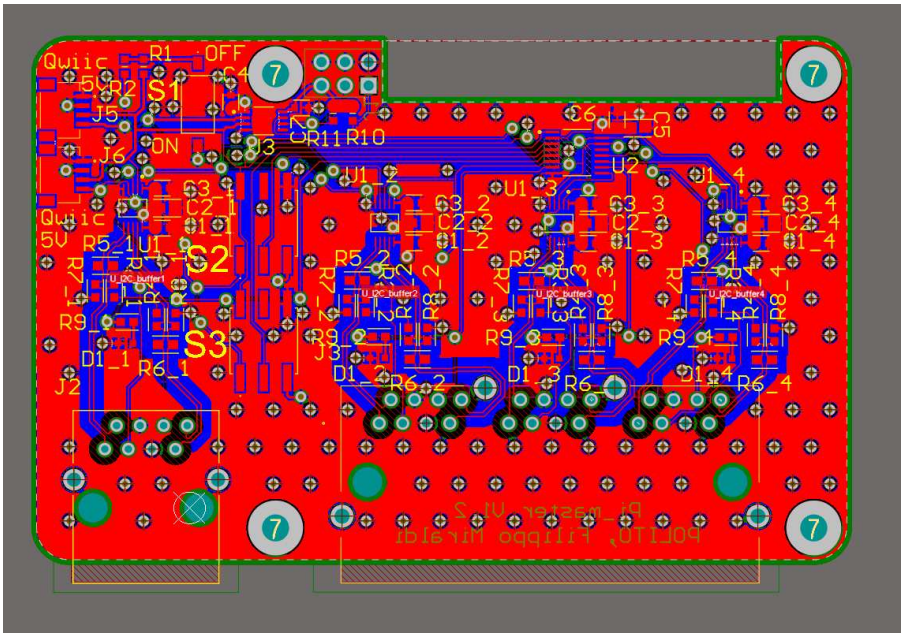
TCA9517 [28] is a small level-shifting I2C buffer IC. It is used to transform the 5V logic level I2C bus used on the board to a 3.3V logic level compatible with the SBC

Resistor R10 and R11 are the two pull-up resistors for the I2C bus at 3.3V. It was calculated that 10kΩ is a value adequate for them.

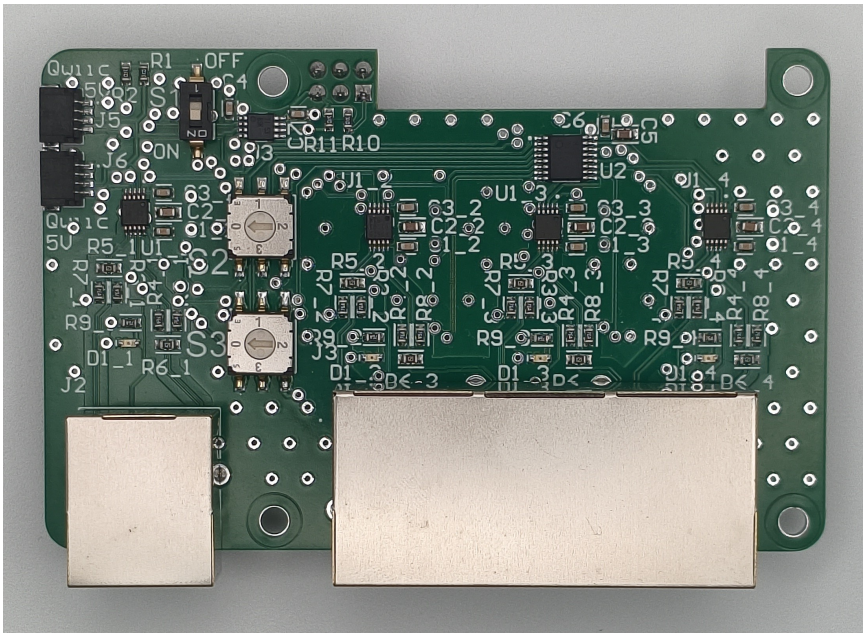
Capacitors C4 and C7 are the two pull-up capacitors.

The 3.3V and 5V power supply for Pi_master and all the accessory and Slave boards is provided by the GPIO header of the Raspberry Pi.

5.2 PCB implementation



(a) Pi_master PCB project file



(b) Pi_master final PCB implementation

Figure 5.5: Side by side view of the project file and the final PCB manufactured

5.2.1 Electrical Design

Figure 5.5 shows the project file with the final manufactured PCB.

The PCB is a four-layer board, 8.7cm x 5.6cm, with the same stackup as the Slave boards, *JLCPCB JLC04161H-7628* (Figure 3.9). It is logically organized as *SIGNAL-GROUND-GROUND-POWER*, meaning that almost all the signals are placed on the top layer, while the bottom layer is occupied by a big power plane carrying the 5V supply for all the components. Having a power plane is advantageous also from the decoupling point of view since it creates a planar capacitor with the inner ground layer.

The top signal layer has been covered with a ground pour and stitching vias.

The layout of the four differential channels is identical across the PCB, and they are placed right behind the corresponding port. All the differential I2C traces are 100Ω controlled impedance traces.

Due to space constraints, the last three channels use a combined three-port RJ45 assembly.

The two black connectors on the top left corner are the two Stemma QT/Qwiic connections.

The IC placed just under the center of the rectangular cutout in the top part of the PCB is MAX7320 I2C port expander.

The IC placed next to the dip-switch is TCA9517 level shifter. It is placed in this corner since it is close to the six-pin connector that interfaces with the Raspberry Pi.

There is no analog section, high-speed electronics, or radio frequency; thus, the layout is more relaxed compared to the Slave boards.

5.2.2 Mechanical Design

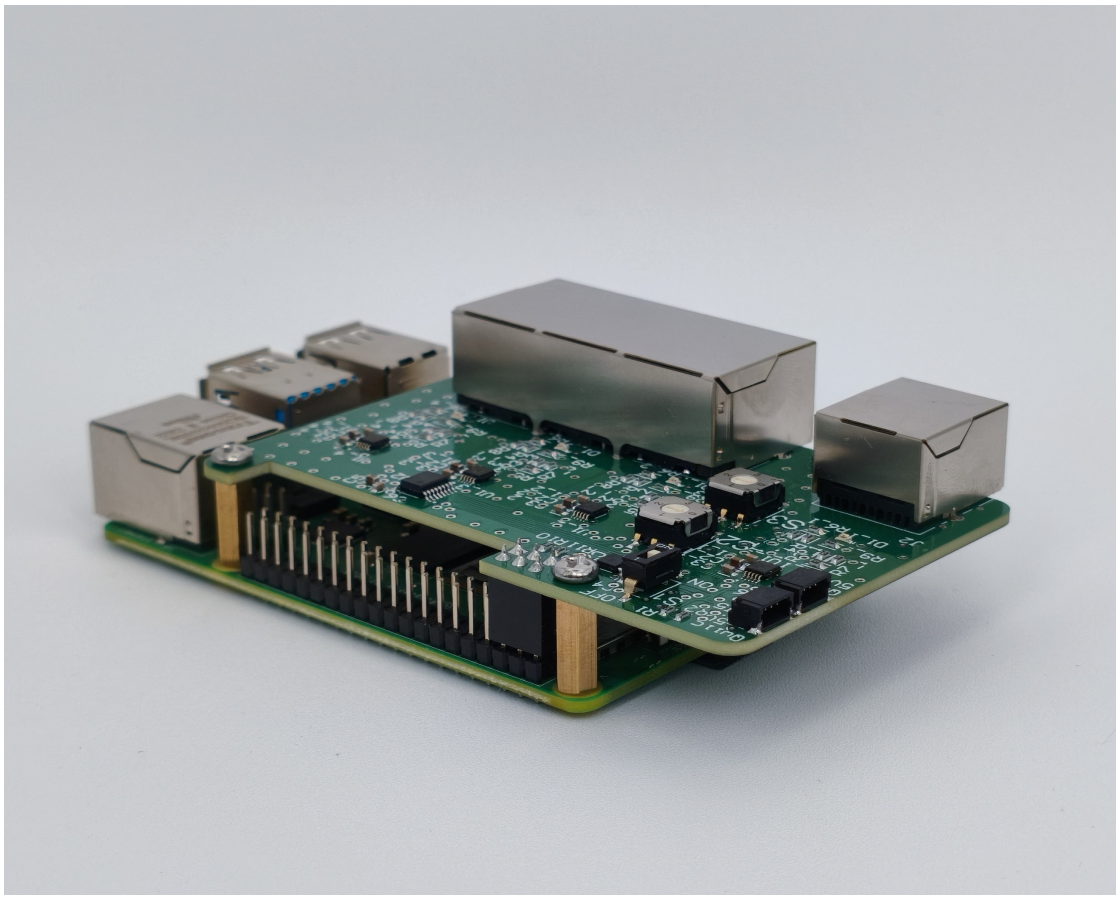


Figure 5.6: Pi_master mounted on the Raspberry Pi

Pi_master is thought to be directly mounted on top of the Raspberry Pi, as shown in Figure 5.6 and Figure 2.14.

Raspberry Pi publishes all the technical drawings and model files of its product, so it was possible to import those files into *Altium Designer 2025* to have the correct position of cutouts and mounting holes to interface with the board.

The PCB does not follow exactly the contour of the Raspberry Pi. It needed to be slightly longer to accommodate the fourth channel and RJ45 port, and has a rectangular cutout over the GPIO pins of the Raspberry Pi to ease possible future modification requiring direct interface with the SBC.

Chapter 6

Optional Accessories Board Hardware

Part of the specification of the project requires the possibility to control small irrigation pumps to selectively and precisely irrigate each pot.

Given the lower complexity of the task, it was chosen to test the modularity of the project using off-the-shelf components and the Stemma QT interface.

A small 3 V submersible pump drawing just 100 mA [29] was selected. (Figure 6.1) placed in a big bucket of water. Each pot has a dedicated tube connected to a dedicated pump.

There is no need to control the speed of the pump since the irrigation is modulated by controlling just the on time of the pump.

A 100 mA load is too much for a single digital pin to handle; an electronic switch is needed. The most flexible and cost-effective solution is represented by a relay board. The used one is shown in Figure 6.2

This board features four reverse logic 5V inputs, each controlling a relay featuring normally closed and normally open contacts. The input circuit is shown in Figure 6.3

The circuit shown in Figure 6.3 is repeated four times, one for each channel.

When the input is low, the current is allowed to flow from the 5V supply through resistor R1 into the optocoupler and through LED D2, used to indicate that the channel is on. The optocoupler, when current passes through its internal LED, allows current to pass and enter transistor Q1 that controls the current through the solenoid of the relay, changing its state. The board features the optocoupler to possibly fully isolate the high voltage side from the low voltage side. The relay can operate at voltages up to 250V AC. For this application, there is no need to have isolation, so the supplies for both sides of the optocoupler are connected using the yellow jumper in Figure 6.2.

Figure 6.1: Small submersible pump used [29]

Diode D1 is important since it protects the circuit from the reverse polarity spike that is generated by the collapsing magnetic field of the solenoid when it is turned off.

To make the relay board compatible with the modular Stemma QT interface used in Pi_master, Adafruit 5346 was used (Figure 6.4). This breakout board features *Microchip MCP23017* 16-channel I/O Expander. This I/O expander gives I2C control to 16 GPIO pins [19].

The Adafruit breakout board features all the necessary components to make the MCP23017 work. Figure 6.5 shows the circuit diagram. LED D1 is a power-on indicator that can be disabled by cutting the trace in the predetermined place (SJ1). Pin A0, A1, and A2 are used to change the address of the device according to Figure 6.6. A small solder bridge can be created on the special pads exposed on the breakout board.

To guarantee better mechanical stability, the breakout board was directly soldered to the relay board. The first and last pins of the control pin header of the relay board, carrying a 5V supply and ground, have been desoldered and substituted by two short wires connected to the power supply and ground pins of the breakout board. Channels 1,2,3, and 4 of the relay board line up with channels A4, A5, A6, and A7 of the breakout board. Figure 6.7 shows the assembled expansion board.

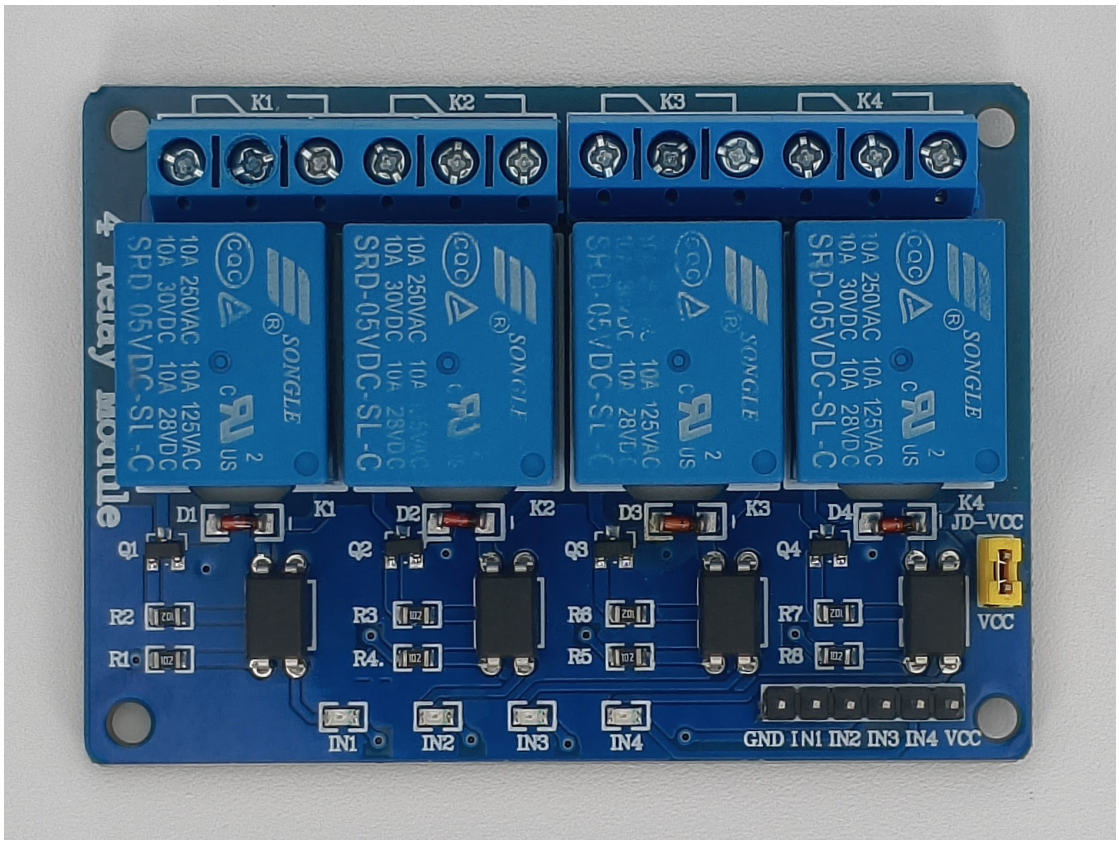


Figure 6.2: Relay board to control pumps

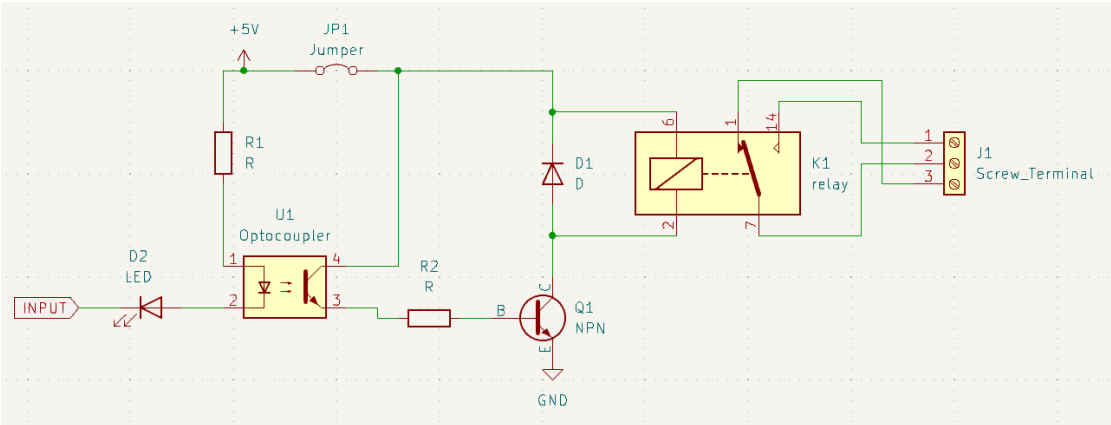


Figure 6.3: Relay board input section

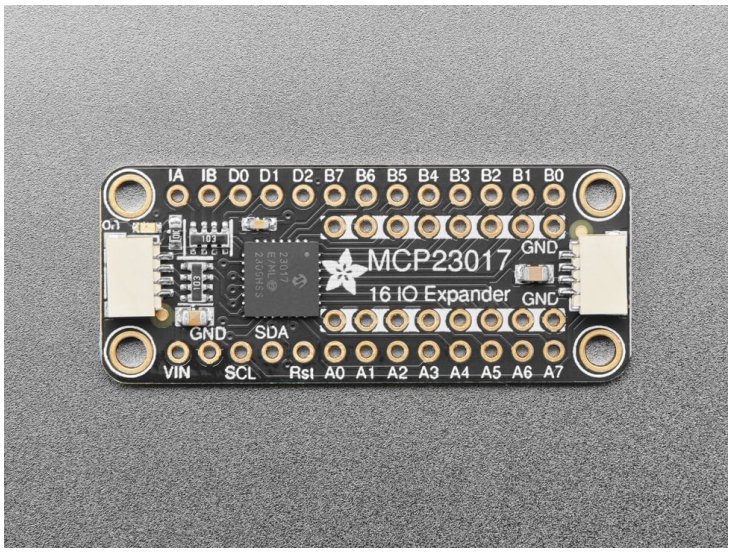


Figure 6.4: Adafruit 5346 I/O expander [30]

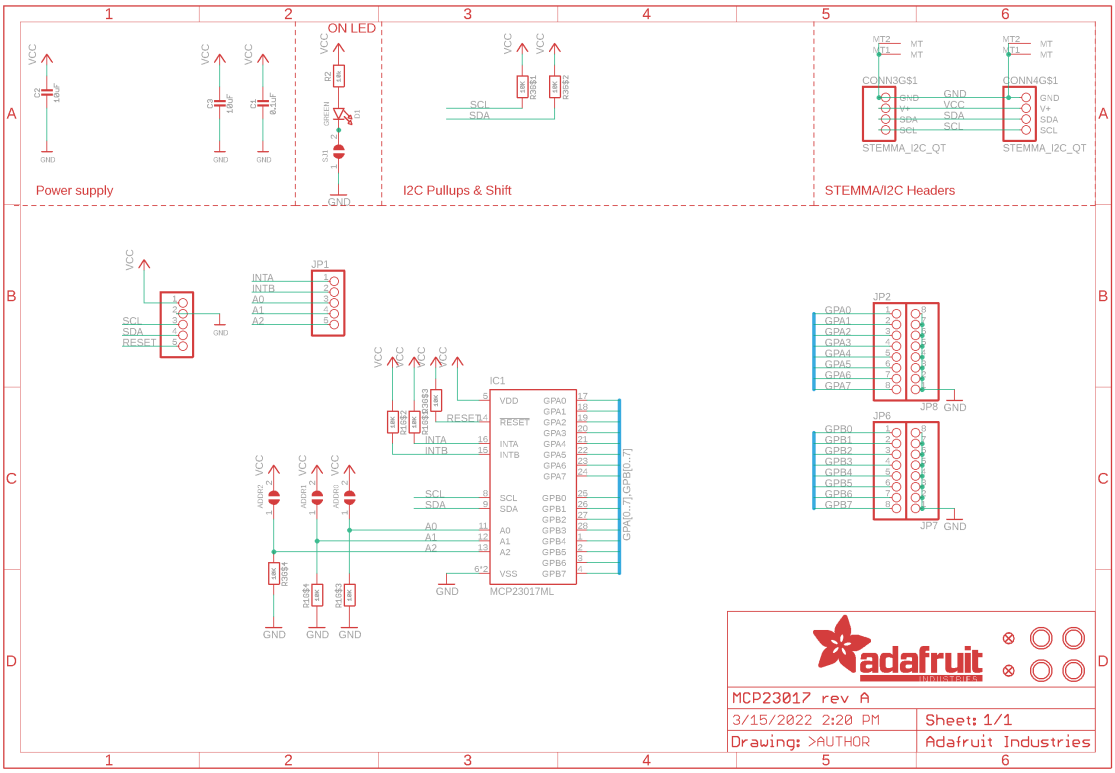


Figure 6.5: Adafruit 5346 circuit diagram [31]

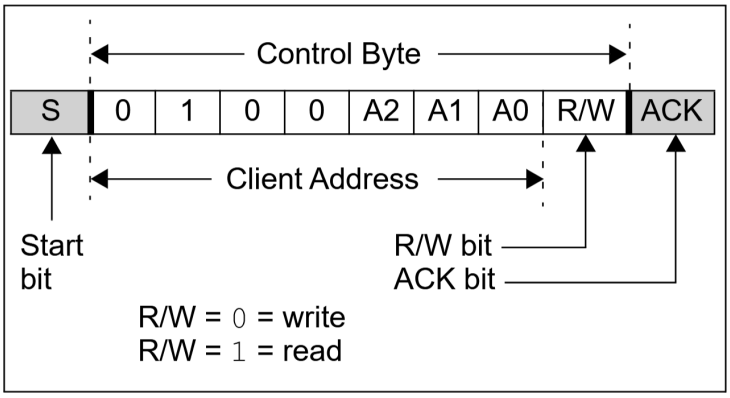


Figure 6.6: MCP23017 I2C adress byte depending on A0,A1 and A2 value

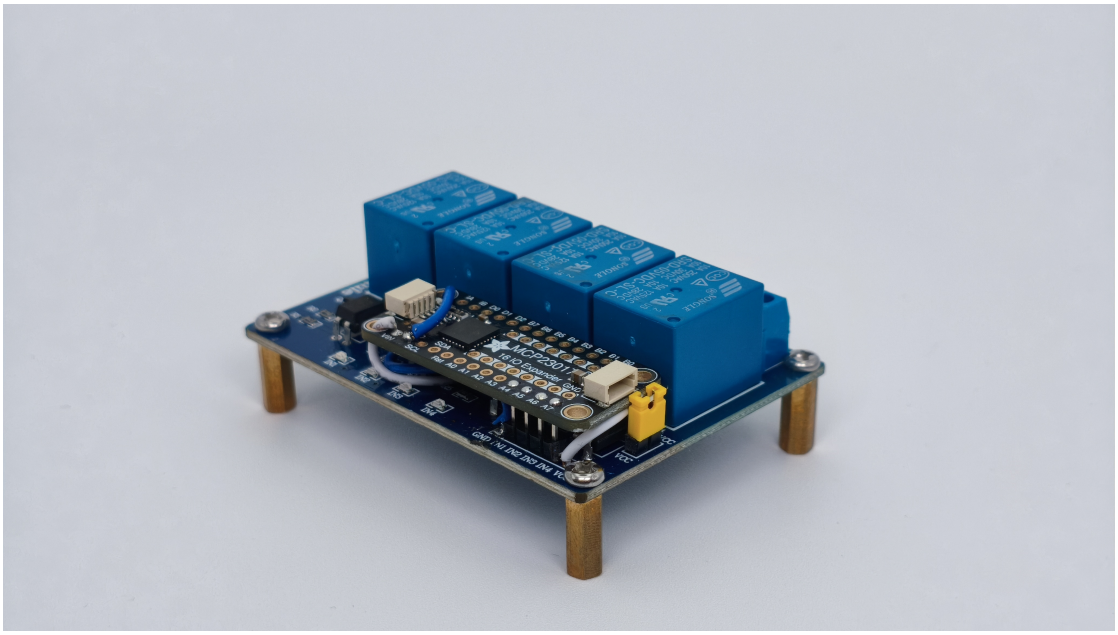


Figure 6.7: Assembled relay expansion board

Chapter 7

Edge Software

7.1 Introduction

The hardware described in the previous chapters requires a digital edge control element that manages the configuration of the various elements, orchestrates the data sampling, and publishes the data. All these functions are performed by the Raspberry Pi 4 model B. This SBC runs Linux, and an ad hoc Python code has been written.

The software has been created with the following objectives in mind:

1. **Plug and play:** sensors can be directly connected, and they should be immediately recognized and start working with no human intervention.

2. **Power outage resilience:** in case power is lost or the computer is rebooted, it should continue working after reboot with no human intervention, including the preservation of calibration coefficients and irrigation schedule.

3. **Ease of installation:** the software must be easy to initialize on the computer with no need to modify the source code.

4. **Ease of maintenance:** since the SBC will be installed headless, commands must be able to be sent to the SBC without the need to directly interact with the machine with mouse and keyboard, everywhere in the world, without an open port or knowing the public or private IP address.

Since the Python project is more than 750 lines of code, only the most essential sections of code will be analyzed in detail, while the rest of the source code is available on the research group's Git-Lab page.

7.2 OS Installation

Raspberry Pi 4 is a normal ARM computer, meaning it is generally impossible to install and use any ARM-based operating system available. With this said, Raspberry Pi OS is a Debian-based, lightweight, 64-bit operating system developed by the manufacturer and optimized for Raspberry Pi boards. This operating system includes a lightweight GUI.

The only non-volatile storage available is the SD card installed on each Raspberry Pi; the operating system has to be installed there.

To install the operating system, it is possible to download the image file from the manufacturer's website and flash it to the SD card using Balena Etcher or Rufus; alternatively, Raspberry Pi offers a dedicated software called *Raspberry Pi Imager* that simplifies the process.

Once downloaded and installed, *Raspberry Pi Imager* opens as shown in Figure 7.1, prompting the user to select the board in use, Raspberry Pi 4 in this case, the OS wanted, this case Raspberry Pi OS, and the SD card where the OS will be installed.

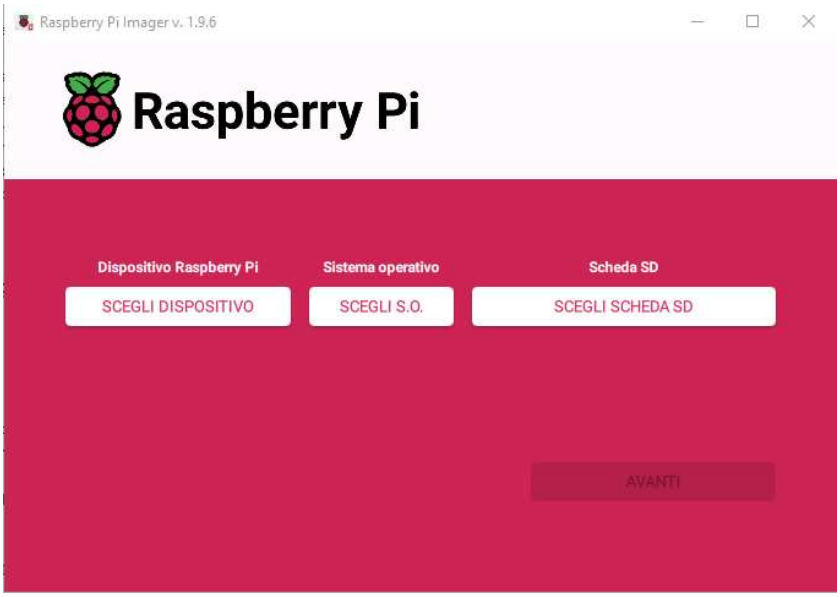


Figure 7.1: *Raspberry Pi Imager* first page view

Once selected and "next" is pressed, a banner appears asking the user if they want to apply operating system customization. (Figure 7.2)

The installation must be customized to guarantee the system works as expected. Once the correct button is pressed, the user is prompted with a view organized into three pages. The two most important ones are shown in Figure 7.3. The last

Figure 7.2: OS customization banner

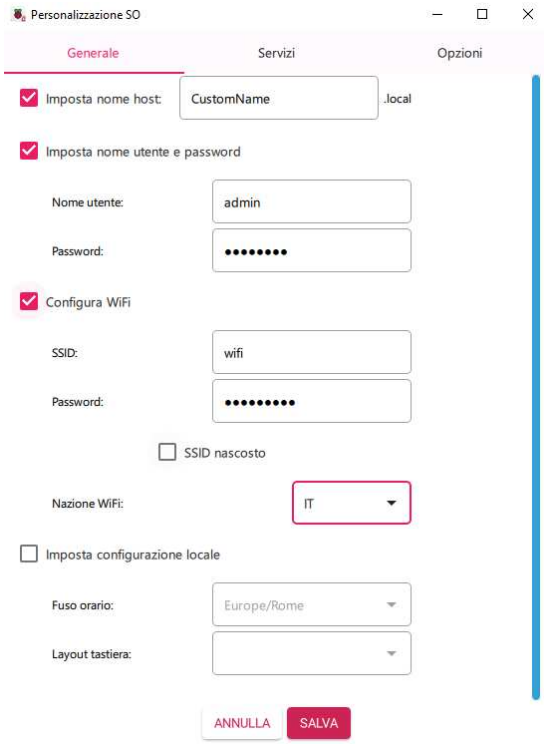
page, named *"options,"* is less important.

On the first page named *"general"*, Figure 7.3a, the hostname must be changed to a unique one. The Python code will automatically tag data with this name to indicate the provenience. It is also advisable to set the user, the password, and the WiFi credentials if it is the planned connection for the specific board. Alternatively, the Raspberry Pi is equipped with a Gigabit Ethernet port, which is the preferred connection method for the greater stability of a wired connection compared to a wireless one.

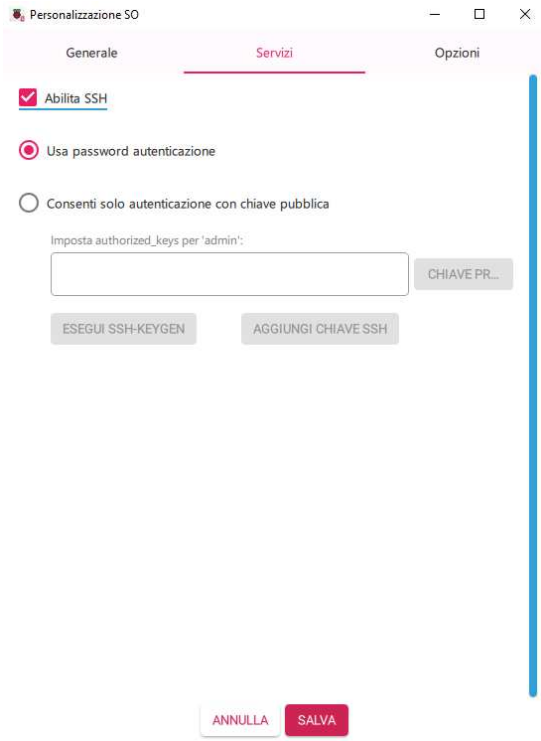
In the second page called *"services"*, Figure 7.3b, it is possible to directly enable SSH to remotely connect to the Raspberry Pi.

If in the local LAN is enabled local DNS, it is possible to connect in SSH to the Raspberry Pi board using the hostname followed by the local domain without the need to set a static IP address or to know the IP address assigned by the local DHCP server. Having already configured the SSH, user, password, and WiFi (if needed), it is possible to complete the setup and use the computer without ever needing to connect a monitor, mouse, or keyboard.

Going back to the OS installation procedure, once the customization is completed, it is possible to proceed. Pi Imager will automatically download the latest version of the operating system and flash it to the SD card.



(a) "General" page



(b) "Services" page

Figure 7.3: Customization menu

7.3 Python Code

The software that manages the acquisition and communication has been written in Python. Python was selected since it is extremely powerful, versatile, and easier to manage, compared to other languages, increasing the ease of customization of the system. The main drawback of Python is its lower performance, which is not impactful since the Raspberry Pi features a microprocessor that is much more powerful than what is strictly needed for the actual requirements of the scale acquisition system.

The code is organized into seven Python files plus one CSV file containing important stored parameters:

- main.py
- channel_select.py
- csv_conf.py
- i2c_bus.py
- irrigation.py
- scales.py
- mqtt_conf.py
- scale_system_calibration_and_data.csv

One extra Python file, set_adress_zssc3224.py, has been written to set the I2C address of the chip zssc3224 initially, and it is reported in Appendix A.

The programming was done in *Microsoft Visual Studio Code* connected in SSH to a headless Raspberry Pi running *Microsoft Visual Studio Code Server*.

7.3.1 main.py

main.py is the first file that is executed at system startup. The first lines executed at boot are reported in Listing 7.1.

```
1  mqtt_connect_with_retry()
2  mqtt_conf.client.loop_start()
3  channel_select.setting_on_boot()
4  scales.setting_on_boot()
5  csv_conf.initialize_csv_file()
6  irrigation.setting_on_boot()
7  print(f"[DEBUG] Waiting 3 seconds before starting")
8  time.sleep(3)
9
10 for fn in (sensor_reader, average_publisher, irrigation_manager):
11     threading.Thread(target=fn, daemon=True).start()
```

Listing 7.1: Startup initialization sequence

The first two lines are used to connect and start the MQTT connection. MQTT is the communication backbone of the application. Every data exiting the system and commands entering pass through MQTT. Function *mqtt_connect_with_retry* is defined in main.py, and it attempts to connect to the MQTT broker and, if the connection fails, repeats the attempt after waiting an increasing amount of time.

Mqtt_conf.client.loop_start() is part of the library *Paho MQTT* that is used to manage the MQTT connections and required to start the service.

Lines 3, 4, 5, and 6, always of Listing 7.1, are function calls to *channel_select.setting_on_boot, scales.setting_on_boot, csv_conf.initialize_csv_file, irrigation.setting_on_boot*. These functions are needed to configure all the different modules at boot, and they will be explained in detail, respectively, in subsections 7.3.3, 7.3.2, 7.3.4 and 7.3.6.

Time.sleep(3) called in line 8 is used to stop the execution of the program for three seconds. It was experimentally seen that the sensors, after the initialization, require a few seconds to be fully ready to read.

Lines 9 and 10 start a thread for each function called *sensor_reader, average_publisher, and irrigation_manager*. Threads are a way for a program to do multiple things at the same time. Standard programs execute instructions sequentially. By using threads, multiple sets of instructions can run concurrently. In this particular case, these sets of instructions are the three functions just listed and represent the essential foundation of the functionality of the software.

Listing 7.2 contains the code executed by the first thread initialized in Listing 7.1. This function reads the scales connected at the RJ45 ports at a constant interval. Readings are performed at a higher frequency compared to data upload.

```
TIME_BETWEEN_WEIGHT_READINGS = 0.1
calibration_lock = threading.Lock()

def sensor_reader():
    while True:
        try:
            with calibration_lock:
                for i in range(4):
                    channel_select.turn_on_channel_n(i)
                    sensor_buffers[i].append(scales.read_to_kg(
    ↪ scales.read_scale(i), i))
            channel_select.turn_off_channels()
        except Exception as e:
            print(f"[ERROR] sensor_reader: {e}")
        time.sleep(TIME_BETWEEN_WEIGHT_READINGS)
```

Listing 7.2: Sensor reader function

The variable *TIME_BETWEEN_WEIGHT_READINGS* is used to set the period, in seconds, between each set of readings. This is a parameter set at design and not meant to be changed by the user. The 0.1 seconds used is the maximum read frequency achievable by both Slaves with the needed settings applied, more details in subsection 7.3.2.

The code block just turns on each channel, performs a reading, saves the reading to the deque *sensor_buffer*, repeating the cycle for all the channels. After all the channels have been read, the thread stops for the pre-set amount of time.

A deque, double-ended queue, is similar to a list, but it is more optimized for fast insertion and deletion both at the beginning and the end.

calibration_lock is used to stop the execution of the thread during a calibration process. This is to avoid multiple threads accessing the same hardware resource at the same time, causing errors.

The second thread is the average publisher code block, shown in Listing 7.3. This code sends the data from the scales to the main cloud server at a time interval defined, in seconds, by the variable *TIME_BETWEEN_WEIGHT_UPLOAD*. This variable is not thought to be modified by the end user, as it is set during the design phase. The upload interval is already faster than required, allowing the final user more freedom to post-process the data.

This block of code is an infinite loop that starts by waiting the set time for the buffer to properly fill through the function *sensor_reader* (Listing 7.2). Once the correct time is passed, it calculates the average of the values in the reading buffer and clears it.

In this program, every time there is an error, the value is not skipped, but it is

```python
TIME_BETWEEN_WEIGHT_UPLOAD = 30

def average_publisher():
    while True:
        try:
            time.sleep(TIME_BETWEEN_WEIGHT_UPLOAD)
            with calibration_lock:

                avgs = [sum(buf)/len(buf) if buf else -99999 for buf in sensor_buffers]
                for buf in sensor_buffers:
                    buf.clear()

                payload = {
                    "timestamp": datetime.now(timezone.utc).isoformat()
                }

                for i, avg in enumerate(avgs):
                    if avg >= -1000:
                        payload[f"average_{i}"] = str(avg)

            mqtt_conf.client.publish(mqtt_conf.TOPIC_AVG, json.dumps(payload))
            print("[PUBLISH] Averages: ", avgs)
        except Exception as e:
            print(f"[ERROR] average_publisher: {e}")
```

Listing 7.3: Average publisher function

returned a very negative value, such as -99999. A reading like this is impossible to be obtained from a scale of this kind, and thus can easily be detected to trigger some remediation strategy in post-processing. In this particular case, if the buffer happens to be empty, -99999 is returned.

The payload, meaning what is transmitted to the main cloud server, is a JSON file containing the timestamp and the averages of all the readings coming from all the non-empty sensor buffers. These averages will be called average_0, average_1, average_2, average_3 according to the channel originating the data.

In case one or more of the scales are not connected, the code will automatically recognize it and will not populate the payload with the data. Since the data is missing from the payload, it will not be written in the database, decreasing the number of false data points collected.

The just-created payload is published via MQTT to the topic defined in the file *mqtt_conf* and analyzed in subsection 7.3.7

```
MAX_IRRIGATION_THREADS = 4

irrigation_schedule = []
irrigation_schedule = csv_conf.load_irrigation_schedule()
irrigation_executor = ThreadPoolExecutor(max_workers=MAX_IRRIGATION_THREADS)
irrigation_lock = threading.Lock()

def irrigation_manager():
    while True:
        try:
            now = datetime.now()
            now_str = now.strftime("%H:%M")
            today = now.strftime("%Y-%m-%d")
            with irrigation_lock:
                for task in irrigation_schedule:
                    if task["time"] == now_str and task.get("last_run") != today:
                        fn = {0: irrigation.irrigate_1, 1: irrigation.irrigate_2, 2: irrigation.irrigate_3, 3: irrigation.irrigate_4}.get(task["sensor"])
                        if fn:
                            irrigation_executor.submit(fn, task["duration"])
                            task["last_run"] = today
            next_check = (now.replace(second=0, microsecond=0) + timedelta(minutes=1))
            sleep_secs = (next_check - datetime.now()).total_seconds()
            time.sleep(max(sleep_secs, 1))
        except Exception as e:
            print(f"[ERROR] irrigation_manager: {e}")
            time.sleep(5)
```

Listing 7.4: Irrigation manager function

The last thread manually defined is the irrigation manager, shown in Listing 7.4. This is another infinite loop that collects the current time and date. Irrigation_lock is similar to the calibration_lock seen in Listing 7.3, and used to prevent any other thread from modifying the irrigation schedule while irrigation is running.

The loop checks if the current time matches the scheduled time for the irrigation and ensures that the irrigation hasn't already run today. The irrigation is defined in the file *scale_system_calibration_and_data.csv*, with more details provided in the subsection 7.3.4. The irrigation schedule is defined as a JSON file containing the time when the irrigation should start, the "sensor", meaning the channel the

schedule refers to, and the duration of the irrigation.

Line 17 selects the right function, between irrigate_1, irrigate_2, irrigate_3, and irrigate_4 in file irrigation.py (analyzed in subsection 7.3.6), to activate the irrigation while line 19 submits the function to the *ThreadPoolExecutor* that runs it in the background without blocking the code execution. This is essential since multiple scheduled irrigations might begin at the same time or operate while another irrigation is already in operation.

The last lines are to limit the execution of *irrigation_manager* to once every minute.

One last extremely important function worth analyzing in *main.py*, called *on_message* and shown in Listing 7.5.

The function *on_message* is essential for the bilateral communication that the software is capable of. In Listing 7.3, the software used MQTT to send messages, while this other function is automatically called every time a MQTT message, from one of the topics the software is subscribed to, is received. The two *if* conditions are used to check if the command is a calibration request or if there is an update to the irrigation schedule. If one of these two requests is recognized, it is automatically called the correct function to handle them.

These are, in the current version of the software, the only two parameters and commands that can be sent from the user.

```python
def on_message(client, userdata, msg):
    global irrigation_schedule
    try:
        payload_str = msg.payload.decode().strip()
        if not payload_str:
            print(f"[WARN] Empty payload from topic {msg.topic}")
            return

        data = json.loads(payload_str)
        print(f"[MQTT] Message from {msg.topic} :", data)

        if msg.topic == mqtt_conf.TOPIC_COMMAND and isinstance(data, dict) and data.get("action") == "calibrate":
            sid = data.get("sensor_id")
            calw = data.get("calibration_weight")
            if sid in {0, 1, 2, 3}:
                threading.Thread(target=calibrate, args=(sid,calw), daemon=True).start()
            else:
                print("[WARN] sensor_id not valid")

        elif msg.topic == mqtt_conf.TOPIC_IRRIGATION:
            if isinstance(data, list):
                with irrigation_lock:
                    irrigation_schedule = [{**t, "last_run": None} for t in data]
                    csv_conf.save_irrigation_schedule(irrigation_schedule)
                print("[MQTT] Planned irrigation updated and saved in the file.")
            else:
                print("[WARN] Payload irrigation is not a list")

    except json.JSONDecodeError as jde:
        print(f"[ERROR] on_message: JSON decode error: {jde} - Payload: {msg.payload}")
    except Exception as e:
        print(f"[ERROR] on_message: {e}")
```

Listing 7.5: Function called on MQTT message reception

7.3.2 scales.py

scales.py is dedicated to all the functions to initialise, perform measurements, and calibrate the scales connected to the RJ45 ports, both in the version using Nau7802 and Zssc3224.

```
port_status = [-99999, -99999, -99999, 99999]

def setting_on_boot():
    for i in range(4):
        channel_select.turn_on_channel_n(i)
        port_status[i] = i2c_bus.i2c_ping()
        set_channel(i)
        if(port_status[i] == 0x2A):
            print(f"[DEBUG] Setting sensor channel {i} with address 0x{port_status[i]:02X}")
            try:
                current_value = bus.read_byte_data(port_status[i], 0x00)
                new_value = current_value
                new_value |= (1 << 7)
                new_value &= ~(1 << 6)
                new_value |= (1 << 2) | (1 << 1)
                bus.write_byte_data(port_status[i], 0x00, new_value)

                current_value = bus.read_byte_data(port_status[i], 0x01)
                new_value = current_value
                new_value &= ~(0b111 << 3)
                new_value |= (0b000 << 3)
                new_value &= ~(0b111)
                new_value |= 0b111
                bus.write_byte_data(port_status[i], 0x01, new_value)

                current_value = bus.read_byte_data(port_status[i], 0x02)
                new_value = current_value
                new_value &= ~(0b111 << 4)
                new_value |= (0b000 << 4)
                new_value |= (1 << 2)
                new_value &= ~(0b11)
                new_value |= 0b00
                bus.write_byte_data(port_status[i], 0x02, new_value)

                current_value = bus.read_byte_data(port_status[i], 0x1B)
```

```
                new_value = current_value
                new_value &= ~(1 << 6)
                new_value |=  (1 << 5)
                new_value &= ~(1 << 4)
                new_value &= ~(1 << 0)
                bus.write_byte_data(port_status[i], 0x1B, new_value)

                current_value = bus.read_byte_data(port_status[i], 0x1C)
                new_value = current_value
                new_value |= (1 << 7)
                bus.write_byte_data(port_status[i], 0x1C, new_value)


                while True:
                    current_value = bus.read_byte_data(port_status[i], 0x02)
                    cal_err = (current_value >> 3) & 0x01
                    if not cal_err:
                        break
                    new_value = current_value | (1 << 2)
                    bus.write_byte_data(port_status[i], 0x02, new_value)
                    time.sleep(0.1)
            except OSError as e:
                print(f"[WARN] I2C write failed for sensor {i}: {e}")
                return False
        elif(port_status[i] == 0x32):
            print(f"[DEBUG] Setting sensor channel {i} with address 0x{port_status[i]:02X}")

            bus.write_byte(0x32, 0x12)
            time.sleep(0.02)

            read = i2c_msg.read(0x32, 3)
            bus.i2c_rdwr(read)

            acc = list(read.buf[:3])
            time.sleep(0.02)

            acc16 = (acc[1] << 8) | acc[2]
            acc16 &= ~(0xFF)
            acc16 |= 0b1000011000111111

            high = (acc16 >> 8) & 0xFF
            low  = acc16 & 0xFF
```

```
            bus.write_i2c_block_data(0x32, 0x52, [high, low])
            time.sleep(0.02)

    channel_select.turn_off_channels()
    return True
```

Listing 7.6: Scales IC setting on boot

Function *setting_on_boot*, shown in Listing 7.6, is used to first initialize all the Slave front ends when the code starts, as shown in Listing 7.1. This function is a cycle that repeats four times, one for each channel. It can be, itself, divided into three different sub-sections:

- fill of *port_status* vector
- eventual initialisation of Nau7802
- eventual initialisation of Zssc3224

port_status is a list of four elements used to automatically recognize the sensor attached to the specific channel. The sensor is recognized by its I2C address. At every execution, the *port_status* value corresponding to the active channel is updated with the actual address. It is initialized at first with an impossible, very negative value that corresponds to a channel with no sensor attached. The address of the attached sensor is obtained with the function *i2c_ping* explained in subsection 7.3.5.

The just collected addresses are then used with a series of *if* statements to selectively initialize each channel in the correct way for the sensor attached.

0x2A is the address of Nau7802.

This sensor presents a series of registers whose values set its behavior. The full explanation and configuration list are available in the datasheet of Nau7802[9]. These registers can be written as an external memory.

The registers are not just rewritten completely, but they get read at first, and on the current state of the register, logic operations are performed to modify exclusively the bits of interest, leaving the rest unchanged to their current value.

Register 0x00 is called *PU_CTRL* and controls the power up of the IC. Bit 7 sets the usage of the internal LDO for the analog supply and not an external analog voltage supply. Bit 6 sets the usage of the internal RC oscillator, rather than an external crystal, for the ADC clock. Finally, bits 1 and 2 power up the analog and digital circuit of the IC

Register 0x01 is called *CTRL1*. Bits 3 to 5 set the LDO voltage. Eight values are available, ranging from 2.4V to 4.5V (see Table 3.1). The selected 000 corresponds to 4.5V, the highest value available. Having higher LDO voltage means higher load cell bridge supply, thus greater output voltage and lower relative effect of noise on

the final output. Bits 0 to 2 set the PGA gain. Eight values are available, from 1 to 128. Since the output from the load cell is very small, it is possible to select the highest gain without risking saturating the input of the ADC.

Register 0x02 is called *CTRL2*. Bits 6 to 4 select the conversion rate. Five values are available from 10SPS to 320SPS. The slower conversion ratio available, 10SPS, was selected since it brought noise advantages compared to the faster ones. Bit 2 triggers a calibration of the sensor, and bits 0 to 1 set the calibration to be internal. This internal calibration is used to correct for unwanted offsets in the reading from internal circuits.

Register 0x1B is called *PGA Registers*. Bit 6 is the LDOMODE; setting it to 0 improves the accuracy at high gains. Bit 5 enables the PGA output buffer, and bit 4 disables the PGA bypass. Bit 0 enables the chopper on the PGA.

The last configured register is register 0x1C, where just bit 7 is set to enable the PGA bypass capacitor (Capacitor C7 in Figure 3.4) that improves performance.

The final part of the configuration of NAU7802 is the loop from line 49 to line 55. It is used to wait until the automatic calibration finishes. If the automatic calibration fails, a new one is started.

Zssc3224 is set with address 0x32 using the code shown in Appendix A. The I2C management of this IC is different compared to the approach used by Nau7802. The configuration is still done by writing particular registers, but the registers are not accessed by their address, but they are accessed using special commands. There is a command to read a specific register and another command to write that register. The datasheet of Zssc3224 [13] offers a complete list of all the registers and the commands to access them. *Commands* are simple I2C writes instructions. The commands to read a register are from 0x00 to 0x39, depending on the specific register. To write a register, the command is the command to read shifted by 0x40, so the commands go from 0x40 to 0x79.

Since the command to access a register is not the address of the register anymore, the registers do not have to be 8 bits long. Zssc3224 uses 16-bit configuration registers.

Code from line 60 to line 79, always from Listing 7.6, configure Zssc3224. Only one 16-bit register is changed. Even though the entire register has been rewritten, the approach with masks, already seen with Nau7802, has been used. Given the peculiarity of the approach used by this IC, the usage of masks gives the opportunity to fully document a full read and write procedure.

With reference to the datasheet of the IC [13], the only register that needs to be changed is *SM_config1*, which configures the measurement. This register is accessed to be read with command 0x12, seen on line 63. With this command, the chip answers with 3 bytes: a status byte followed by two bytes with the contents of the register (line 69). The register write procedure is seen in line 78. Please note that register write is not performed with command 0x12 anymore, but it is 0x52

since the read and write commands have an offset of 0x40.

The value written in the register is *0b1000011000111111*. Starting from the right, the first three bits *111* set the first PGA gain to 120, bits 3 to 5 set the second PGA gain to 1.8. In both cases, there are a total of eight different gain alternatives, and, for this application, both were set to the maximum gain available. Bit 6 is to invert the polarity of the input. 0 is with no polarity change. Bits 7 to 10 set the number of bits of the ADC. The minimum number of bits is 12, and the maximum is 24. 1100 sets it to 24 bits. Bit 11 is to enable the thermopile input, not used in this application, and so is set to 0. Bits 12 to 14 are to compensate offset on the signal. It is not used. The last bit on the left is set to 1 and enables a shift on the ADC reading of one, giving an extra equivalent gain of 2.

Once the configuration for a channel is completed, it is disabled, and the code continues with the configuration of the rest of the channels.

The function shown in Listing 7.7 reads the raw value from the ADCs. This function is called every time the raw ADC reading is required.

Analogously to the function *setting_on_boot* shown in Listing 7.6, even this last code block is divided into two different parts: one for Zssc3224 and one for Nau7802. The list *port_status* filled during the initialization of the scales is used to know what sensor is connected to what channel.

The read for Nau7802 (sensor 0x2A) starts at line 4 of Listing 7.7. There is an infinite *while* loop that is used to wait for bit 5 of register 0x00 to become 1. This bit is the *Cycle ready* bit, indicating when the ADC data is ready to be read. Once this bit is equal to 1, the cycle can exit.

The 24-bit reading is divided into registers 0x12, 0x13, and 0x14. These values have to be concatenated to compose a single 24-bit number.

For Zss3224 the command 0xA2 is used to obtain what the datasheet defines as *"24-bit formatted raw data, Auto-Zero corrected Sensor Measurement"* [13]. The command answers with four bytes, since the first bit is always a status byte.

Zssc3224 offers extensive and complex integrated DSP functionalities capable of correcting sensor non-linearities due to construction and temperature. This functionality is not used in this project since the manufacturer of the load cells does not provide enough information to properly set the DSP up.

Once the raw data is obtained using the function *read_to_kg*, the raw ADC data is converted to a kg reading using Equation 3.3. The function *cal_read_to_kg* takes the kg reading just obtained and applies a linear calibration using the gain and bias parameters, specific for the channel, saved in the CSV file.

The parameters saved in the CSV file are calculated using the last function in *scale.py*: *calibrate* (Listing 7.8)

Since there is a need to calculate two different parameters, it is necessary to calibrate with two different weights. The first weight selected is 0kg, meaning the empty scale plate. This provides an immediate indication of the offset of

```python
def read_scale(sensor_id):
    weight = [0, 0, 0]

    if(port_status[sensor_id] == 0x2A):
        while True:
            status = bus.read_byte_data(port_status[sensor_id], 0x00)
            if status & (1 << 5):
                break
            time.sleep(0.01)
        weight[0] = bus.read_byte_data(port_status[sensor_id], 0x12)
        weight[1] = bus.read_byte_data(port_status[sensor_id], 0x13)
        weight[2] = bus.read_byte_data(port_status[sensor_id], 0x14)
        weight_value = (weight[0] << 16) | (weight[1] << 8) | weight[2]
        return weight_value

    elif(port_status[sensor_id] == 0x32):
        res = bus.read_i2c_block_data(port_status[sensor_id], 0xA2, 4)
        time.sleep(0.02)
        weight_value = (res[1] << 16) | (res[2] << 8) | res[3]
        return weight_value

    else:
        return -99999
```

Listing 7.7: Raw ADC value reading

the measurement caused by load cell residual unbalance, by non-idealities of construction, and by the weight of the structure composing the plate of the scale. The second weight is set by the user. When the user wants to start a new calibration process on a particular channel, they must send a JSON file (see Appendix B) to the correct MQTT topic explained in subsection 7.3.7, containing also the channel number and the exact calibration weight used. This allows us to place the second calibration point as close as possible to the average expected weight, guaranteeing greater accuracy.

The calibration procedure communicates with the user, sending messages through a dedicated MQTT topic. The communication happens with an MQTT quality of service of 2 to guarantee the receipt of the message, see subsection 2.4 for more details.

Once the calibration request message is received, the program communicates

to the user that they have 10 seconds to remove everything from the platter of the scale. Once the 10 seconds have elapsed, the program performs 1024 readings from the scale using the function *read_scale*. The number of readings is set by the variable *CALIBRATION_AVERAGES*, set at the design process. The readings are then averaged and saved. Afterwards, the program communicates to the user that they have 30 seconds to place the calibration weight on the scale. After the timer elapses, the 1024 readings are repeated, and a second average value is stored.

The offset parameter for the linear calibration is directly obtained from the zero load reading, while the gain parameter is obtained in line 26 of Listing 7.8. The two final values are finally written to the CSV file.

```
CALIBRATION_AVERAGES = 1024
def calibrate(sensor_id, calibration_weight):
    info = mqtt_conf.client.publish(mqtt_conf.TOPIC_COMMUNICATION, json.dumps({"status": f"Remove all loads from scale {sensor_id}. You have 10 seconds"}), qos=2)
    info.wait_for_publish()
    time.sleep(10)
    info = mqtt_conf.client.publish(mqtt_conf.TOPIC_COMMUNICATION, json.dumps({"status": f"Starting step one of the procedure, it will take {CALIBRATION_AVERAGES/10}s"}), qos=2)
    info.wait_for_publish()
    i = 0
    sum = 0
    for i in range(CALIBRATION_AVERAGES):
        sum = sum + read_scale(sensor_id)
        time.sleep(0.1)
    avg_0 = sum/CALIBRATION_AVERAGES
    print(f"[CALIBRATE] Avg raw offset: {avg_0}")
    avg_0_kg = cal_read_to_kg(avg_0, sensor_id)
    info = mqtt_conf.client.publish(mqtt_conf.TOPIC_COMMUNICATION, json.dumps({"status": f"Put on scale {sensor_id} the calibration weight of {calibration_weight}Kg . You have 30 seconds"}), qos=2)
    info.wait_for_publish()
    time.sleep(30)
    info = mqtt_conf.client.publish(mqtt_conf.TOPIC_COMMUNICATION, json.dumps({"status": f"Starting step two of the procedure, it will take {CALIBRATION_AVERAGES/10}s"}), qos=2)
    info.wait_for_publish()
    i = 0
    sum = 0
    for i in range(CALIBRATION_AVERAGES):
        sum = sum + cal_read_to_kg(read_scale(sensor_id), sensor_id)
        time.sleep(0.1)
    avg_slope = calibration_weight/((sum/CALIBRATION_AVERAGES)-avg_0_kg)
    print(f"[CALIBRATE] Avg slope: {avg_slope}")
    csv_conf.update_calibration_value(sensor_id, avg_0_kg, avg_slope)
    info = mqtt_conf.client.publish(mqtt_conf.TOPIC_COMMUNICATION, json.dumps({"status": f"Offset channel {sensor_id} updated to {avg_0_kg} and slope updated to {avg_slope}"}), qos=2)
    info.wait_for_publish()
    time.sleep(2)
    return
```

Listing 7.8: Channel calibration function

7.3.3 channel_select.py

Channel_select.py contains the functions that control MAX7320, the I2C port expander used to multiplex the RJ45 ports used to connect the scales.

```
adress = 0b1011101

def turn_on_channel_n(channel_id):

    if(channel_id == 0):
        bus.write_byte(adress, 0b00000011)

    elif(channel_id == 1):
        bus.write_byte(adress, 0b00001100)

    elif(channel_id == 2):
        bus.write_byte(adress, 0b00110000)

    elif(channel_id == 3):
        bus.write_byte(adress, 0b11000000)
    else:
        print(f"[ERROR] scale channel selected out of range")
```

Listing 7.9: Function to activate a specific scale channel

MAX7320 is very simple to control. Listing 7.9 is the function used to activate a specific channel. When the function is called, it is also passed the channel number that is intended to be turned on. Every I2C message has a payload of eight bits, the same number of ports available in MAX7320. Each bit of the message sent to the IC directly controls one port. So, for example, to turn on just the first channel of the IC bit zero of the sent message should be 1, while the rest of the bits should be 0. As explained in subsection 5.1.2, MAX7320 not only multiplexes a channel but also independently controls an LED, which in this application is tied whit the activation of a specific differential I2C bus. Two pins of the IC have to be turned when a single channel is enabled. Depending on the specific channel that is needed, function *turn_on_channel_n* sends an adequate I2C message to enable the specific ports tied to the enable pin of the channel and to the correct LED.

Channel_select.py also contains the function *turn_off_channels* that simply sends a payload of 0x00 to disable all the channels and the function *setting_on_boot* that calls the *turn_off_channels* to start the code with all channels disabled.

For the time being, the code expects MAX7320 to have an address of 0b1011101. The hardware is already prepared for future software versions, where multiple sensors might be chained together, and the MAX7320 address can be easily changed using the on-board rotary selection switches.

7.3.4 csv_conf.py and scale_system_calibration_and_data.csv

Csv_conf.py contains all the functions needed to read and update the .csv file. The .csv file, *scale_system_calibration_and_data.csv*, is used to store the calibration parameters and the irrigation schedule in a non-volatile manner. The usage of this file allows the system to recover from a power cycle without human intervention or the need for a lengthy calibration procedure.

The first function that is present is *initialize_csv_file*. This function checks if the .csv file is present and not empty. If the file were to be found empty or not present, the function creates a new file with standard values as shown in Listing 7.10

```
1 0,0,0,0
2 1,1,1,1
3
4 "[{""time"": ""08:00"", ""sensor"": 1, ""duration"": 180, ""last_run"": null},
    ↪ {""time"": ""08:00"", ""sensor"": 2, ""duration"": 180, ""last_run"":
    ↪ null}, {""time"": ""08:00"", ""sensor"": 3, ""duration"": 180, ""
    ↪ last_run"": null}, {""time"": ""08:00"", ""sensor"": 4, ""duration"": 18
    ↪ 0, ""last_run"": null}]"
```

Listing 7.10: Default scale_system_calibration_and_data.csv

Line 1 contains the offset parameter. It is initialised at zero when no calibration is set. Line 2 is the gain calibration parameter, initialized at 1. Line 4 contains the four JSON files with the irrigation program. Each column of the .csv file corresponds to a specific channel.

Update_calibration_value is the function used to save the calibration parameter generated by the calibration procedure, see Listing 7.8, while *get_zero_calibration_value* and *get_slope_calibration_value* are used to retrieve the calibration values.

Finally, *load_irrigation_schedule* and *save_irrigation_schedule* are used to retrieve the irrigation schedule from the file and to load an updated irrigation schedule.

7.3.5 i2c_bus.py

```
from smbus2 import SMBus

bus = SMBus(1)

def i2c_ping():
    addresses_to_check = [0x2A, 0x32]
    for addr in addresses_to_check:
        try:
            bus.read_byte(addr)
            return addr
        except OSError:
            continue
    return -99999
```

Listing 7.11: i2c_bus.py complete file

Listing 7.11 reports the file *i2c_bus.py* in its entirety.

Funcion *i2c_ping* checks if any device with an address present in the list *addresses_to_check* is connected to the bus. To do so, it attempts to read an inexistent message from each device address, proceeding in order from the first to the last. If the command reports an error, it means that the device is not connected or not responding. If no error arises means that a device with that address is connected to the bus and that specific address is returned.

Since 0x2A is the address of Nau7802 and 0x32 is the address selected for Zssc3224, and just one of these two devices can be connected to a single RJ45 interface, the search can stop at the first answer.

If all the addresses in *addresses_to_check* raise an error, it means that no functioning sensor is connected to the port, and a very negative value is returned to indicate the condition.

The *i2c_ping* function is key in the procedure of automatic recognition of the connected sensor.

7.3.6 irrigation.py

Irrigation.py contains all the functions that are directly related to the active control of the irrigation board. The first function is the *setting_on_boot* used in the *main.py*, see Listing 7.1.

```
def setting_on_boot():
    bus.write_byte_data(0x20, 0x00, 0x00)
    bus.write_byte_data(0x20, 0x12, 0xFF)
```

Listing 7.12: Irrigation setting_on_boot function

Listing 7.12 shows the complete function. Two I2C write operations are performed. As explained in chapter 6 key in the I2C control of the relay is the MCP23017 I/O expander.

It is set to its base address of 0x20.

Given its ability to perform both input and output from its 16 digital pins, this IC is characterized by a greater complexity compared to MAX7320 previously analyzed.

The first command in line two is an I2C write to set the pins of interest to output mode, while the second command sets all the pins to high. Pins need to be high to turn off the pump because the relay board's inputs are in reverse logic.

Other than *settin_on_boot* , *irrigation.py* contains other four almost identical functions called *irrigate_1, irrigate_2, irrigate_3* and *irrigate_4*. These functions only differ in the value of the variable *bit*. *irrigate_1* is shown in Listing 7.13.

```
def irrigate_1(duration):
    bit = 0b00010000
    print(f"[IRRIGATE] Zone 0 : starting for {duration}s")
    acc = bus.read_byte_data(0x20, 0x14)
    acc = acc & ~bit
    bus.write_byte_data(0x20, 0x14, acc)
    time.sleep(duration)
    acc = bus.read_byte_data(0x20, 0x14)
    acc = acc | bit
    bus.write_byte_data(0x20, 0x14, acc)
    print(f"[IRRIGATE] Zone 0 : stopped after {duration}s")
```

Listing 7.13: Irrigate_1 function

The variable *bit* represents the bit that needs to be activated to enable the specific relay tied to the specific channel number. Afterwards, using the mask procedure, this bit is sent to the MCP23017 without changing the value of the other bits in the register.

As seen in Listing 7.4, a new thread is created when each of these functions is called. A sleep function can be used to set the duration of the irrigation. Once the thread wakes back up, the irrigation can be stopped using a similar methodology seen before.

It is important not to change the value of the rest of the register since multiple irrigations with different lengths might be active at the same time.

7.3.7 mqtt_conf.py

The last file is *mqtt_conf.py*, reported in its completeness in Listing 7.14.

```
import socket
import paho.mqtt.client as mqtt

HOSTNAME = socket.gethostname()

TOPIC_AVG = f"scales/{HOSTNAME}/sensors/average"
TOPIC_COMMAND = f"scales/{HOSTNAME}/device/commands"
TOPIC_IRRIGATION = f"scales/{HOSTNAME}/device/irrigation"
TOPIC_COMMUNICATION = f"scales/{HOSTNAME}/device/communication"

client = mqtt.Client(protocol=mqtt.MQTTv5)
```

Listing 7.14: mqtt_conf.py complete file

This file only contains the MQTT topics that are used by the entire program. As explained in section 2.4, topics are hierarchical. All the messages that are sent or received by the system are under the general topic *scale/* and under the subtopic *HOSTNAME/*. *HOSTNAME/* is automatically substituted by the computer hostname, set at the OS installation phase (section 7.2). This is done using the Python socket library.

This approach allows automatic tagging of the messages from a particular node using the node name itself, and with no need to modify the Python code directly.

Sub-sub topics are two. *Sensors/* is used for the values read by the sensors mounted, while */device* is used for general communication to the user, setting up the irrigation schedule, or sending commands, such as the calibration start procedure.

Please see Appendix B to see examples on how to send calibration and irrigation commands.

/sensors/average is used to transmit the JSON file containing the readings from the scales.

7.4 Python-OS Interaction and Software Setup

The Python project explained in section 7.3 has to run every time the computer starts and continues running until the computer is stopped. To achieve this, a dedicated Linux service was created. A service is a background process that performs a specific task without requiring direct user interaction.

To simplify the deployment, a configuration Bash script was written. (Listing 7.15).

This script, called *python-scales-installer.sh*, is saved on a USB stick with a folder called *python-scales-project*, containing all the source files of the Python project.

Once the USB stick is inserted, the script can be run, and the computer is automatically configured.

Listing 7.15: file python-scales-installer.sh

```
set -euo pipefail

# require root
if [[  "$EUID" != "0" ]]; then
  echo "Run with sudo: sudo \"$0\""
  exit
fi

# directory containing this script
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# source project folder
SRC_DIR="$SCRIPT_DIR/python-scales-project"
if [[ ! -d "$SRC_DIR" ]]; then
  echo "Source folder not found: $SRC_DIR"
  exit
fi

TARGET_DIR="/opt/python-scales-project"
SERVICE_NAME="python-scales.service"
MAIN_PY="$TARGET_DIR/main.py"

echo "Installing from: $SRC_DIR"
echo "To: $TARGET_DIR"
echo

# enable I2C if missing
if ! grep -Eiq '^\s*dtparam=i2c_arm=on' /boot/firmware/config.txt; then
  echo "" >> /boot/firmware/config.txt
```

```
  echo "# enabled by python-scales-installer" >> /boot/firmware/config.txt
  echo "dtparam=i2c_arm=on" >> /boot/firmware/config.txt
  NEED_REBOOT=1
else
  NEED_REBOOT=0
fi

# install system packages
apt-get update -y
apt-get install -y i2c-tools python3-venv || echo "apt install failed, install manually"

# copy project to /opt
mkdir -p "$TARGET_DIR"
rsync -a "$SRC_DIR"/ "$TARGET_DIR"/

# create venv and install Python modules
python3 -m venv "${TARGET_DIR}/venv"
"${TARGET_DIR}/venv/bin/pip" install --upgrade pip
"${TARGET_DIR}/venv/bin/pip" install paho-mqtt smbus2 || echo "venv pip install failed, install paho-mqtt/smbus2 manually"

# create systemd service
SERVICE_FILE="/etc/systemd/system/${SERVICE_NAME}"

echo "[Unit]"                        >  "$SERVICE_FILE"
echo "Description=Python Scales Project (main.py)"   >> "$SERVICE_FILE"
echo "After=network-online.target"                   >> "$SERVICE_FILE"
echo                                 >> "$SERVICE_FILE"
echo "[Service]"                     >> "$SERVICE_FILE"
echo "Type=simple"                   >> "$SERVICE_FILE"
echo "WorkingDirectory=${TARGET_DIR}"                >> "$SERVICE_FILE"
# use the venv python
echo "ExecStart=${TARGET_DIR}/venv/bin/python ${MAIN_PY}" >> "$SERVICE_FILE"
echo "Restart=always"                >> "$SERVICE_FILE"
echo "RestartSec=5"                  >> "$SERVICE_FILE"
echo                                 >> "$SERVICE_FILE"
echo "StandardOutput=journal"        >> "$SERVICE_FILE"
echo "StandardError=journal"         >> "$SERVICE_FILE"
```

```
67 echo                                >> "$SERVICE_FILE"
68 echo "Environment="PYTHONUNBUFFERED=1""                                >>
    ↪ "$SERVICE_FILE"
69 echo                               >> "$SERVICE_FILE"
70 echo "[Install]"                              >> "$SERVICE_FILE"
71 echo "WantedBy=multi-user.target"                              >>
    ↪ "$SERVICE_FILE"
72
73 systemctl daemon-reload
74 systemctl enable ${SERVICE_NAME}
75 systemctl restart ${SERVICE_NAME} || echo "Service created but failed
    ↪ to start"
76
77 echo
78 echo "Done. Project installed to: $TARGET_DIR"
79 echo "Service: ${SERVICE_NAME} (enabled)."
80 if [[ $NEED_REBOOT -eq 1 ]]; then
81   echo
82   echo "I2C was just enabled, REBOOT for it to take effect"
83 fi
```

Starting from line 1 of Listing 7.15, "set -euo pipefail" is for safety. This is used to gracefully exit the script in case any error might occur.

Lines from 4 to 7 are used to check if the script has been properly started using sudo, which grants sufficient permissions to successfully execute some of the commands required.

Line 10 assigns the directory of the script to the variable *SCRIPT_DIR*.

Lines from 13 to 17 check if the folder *python-scales-project* containing the source files of the Python project is present.

Lines from 28 to 35 check if the line enabling I2C is present in */boot/firmware/-config.txt*. If it is not present, it adds it.

Lines 38 and 39 install "i2c-tools" and "ptyhon3-venv". "i2c-tools" is a collection of tools to use I2C, while "python3-venv" is necessary to create Python virtual environments. If the command fails, the user is prompted to install the tools manually.

In lines 42 and 43, a folder is created in */opt*, the Linux folder for optional software, and then the project files are moved from the USB stick to a folder on the Raspberry Pi. The folder location is set in line 19. "rsync" is used to synchronize the USB stick folder to one on the local machine. It was selected over "cp" because this allows the script to also be used for updates to the software.

Lines from 46 to 48 create the "venv" virtual environment for Python and install the required packages.

Lines from 51 to 71 create the *systemd.service* that is a configuration file that

configures the service created to run the Python project. Lines 54 and 55 define the name of the service and set it to start after "network-online.target", ensuring the network is fully up when the service starts.

Lines from 57 to 61 are used to actually start the execution of the Python project.

Lines 62 and 63 set the service to always restart if it is stopped and to just wait 50 seconds before attempting to restart to avoid excessive load on resources.

Lines from 65 to 68 are used to set the code to print to the journal, the service log, and all the outputs of the code.

Line 71 is used to set the service to run at boot.

Lines from 73 to 75 enable the service.

Appendix C explains how to properly use the script to set up a new node.

Chapter 8

Server

Each sensor data aggregator collects the readings from each scale and publishes them on the MQTT topic *scales/{HOSTNAME}/sensors/average*, where *HOSTNAME* is the name of the specific SBC sending the message.

MQTT messages are, generally, volatile by nature; if no devices are listening at the moment the message is sent, it is lost forever. It is thus necessary to have a system always available that is capable of receiving the generated data, managing it, securely storing it, and, if necessary, processing it.

Figure 8.1 gives a schematic view of the server architecture used.

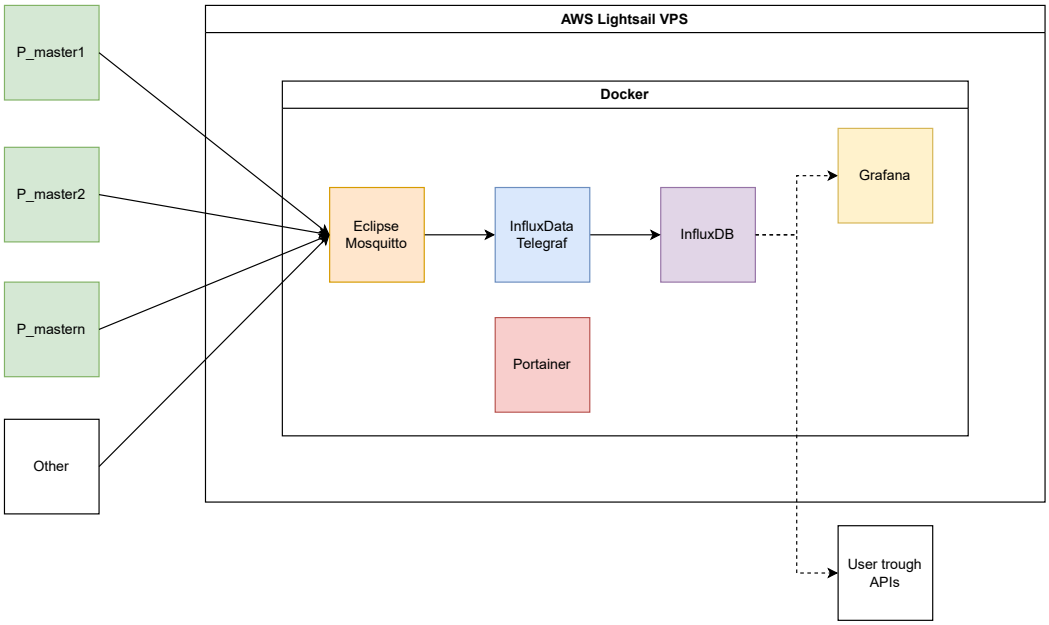


Figure 8.1: Server system architecture block diagram

8.1 Amazon Lightsail VPS

The basis of a server is the machine on which the services run. The *eLiONS* research group has already set up an *Amazon Lightsail VPS*, currently used for other monitoring projects.

A virtual private server is a type of web hosting service where a portion of a powerful server, utilizing virtualization technology, is rented to clients for a monthly fee. Each rented instance behaves as if it were a dedicated server with its own dedicated cores, RAM, storage, and operating system.

Amazon Web Services provides two cloud server solutions offering VPS: EC2 and Lightsail. Lightsail is a less complex and less expensive solution more adequate for simpler hosting applications like this one.

Different tiers of Lightsail differ in the number of dedicated CPU cores, amount of RAM, and amount of available storage. The *eLiONS* machine has 2 cores, 4 GB of RAM, and 80 GB of storage. Each Lightsail instance includes a fixed public IP address, IPv4 for the *eLiONS* machine, and a virtual firewall.

In this machine was installed Debian, a free and open-source operating system based on the Linux kernel. Debian is one of the most common and oldest Linux distributions, characterized by high stability. Debian is the basis on which many other operating systems, such as Ubuntu and Raspberry Pi OS, are built.

Figure 8.2: Amazon Lightsail Logo

8.2 Containerization and Docker

The Lightsail server used to use a traditional approach to software deployment. Every software tool was installed directly on the host server. This could be problematic in some situations.

Applications need dependencies that have to be installed alongside the main application. Different software needs different dependencies, sometimes different versions of the same dependencies, that might conflict with each other, causing instabilities or breakages.

The strong coupling of the installed application with the OS means that new OS versions and updates might cause applications to stop working.

Some security concerns are particularly relevant for a server that is directly exposed to the internet. Installing many applications on the same host's OS increases the attack surface. If a single application is compromised might be easier for the attacker to move laterally and compromise other parts of the system unless proper hardening is applied.

One possible solution to solve some of these problems is the use of virtualization. Virtualization is a software layer of abstraction that allows the creation of multiple virtual computers coexisting on the same physical hardware. Each virtual computer, VM, is completely independent from the others with its own resources (CPU, RAM, ...) and operating system. It is the same principle used by Amazon to create the VPS.

Even though it is possible and commonly used, virtualization is not a good solution for this specific application. First of all, a dedicated VM would have to be created for each software application to have full isolation, each with its own operating system and dedicated hardware resources. This would solve the possible issue with dependencies and the safety concern, since every application would effectively be installed on a separate and independent machine.

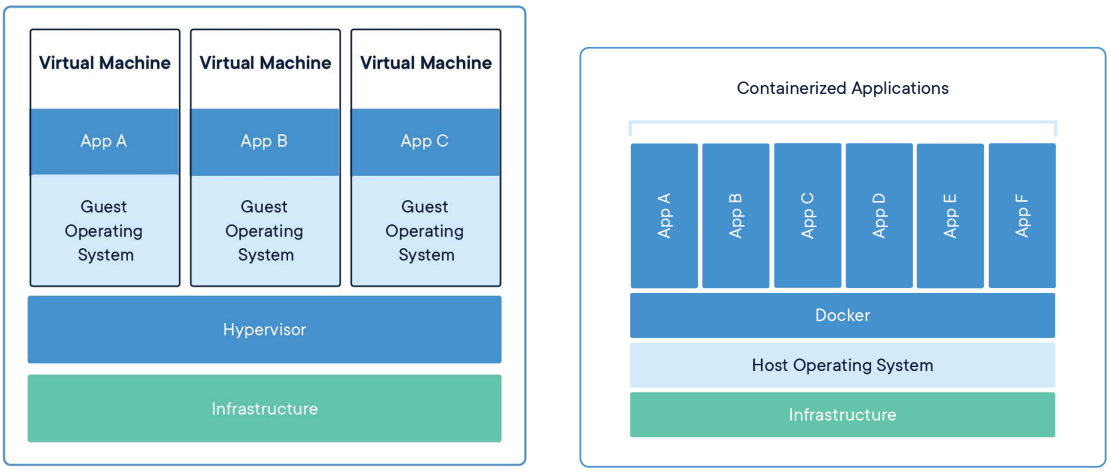
Each VM being fully software-defined, is easy to back up and restore quickly and with ease in case problems would arise.

Using VM would not help with the issue regarding OS updates and would increase the management complexity and the overhead burden on computer resources, since now multiple operating systems would have to run at the same time, an impossible task for a machine with resources as limited as the Lightsail VPS used.

Containerization represents the perfect in-between solution.

Containers package an application as a single portable image of software containing all the configuration files, dependencies, and libraries required to run it. This package is called image. Unlike a virtual machine, a full OS is not bundled in, and the container shares the same kernel as the underlying operating system through an isolation layer provided by the container runtime and managed through a container engine, of which Docker is the industry standard.

Each container appears abstracted away from the host operating system, providing portability of the software and the ability to seamlessly move the application between different machines.



(a) VM stack **(b)** Docker stack

Figure 8.3: Comparison between container-based and VM-based architecture.

From the security perspective, containers provide a much greater isolation compared to the monolithic approach, where every app is directly installed on the host's OS, but at the same time, it is not a complete isolation like the one provided by a VM, leaving the system still exposed to possible kernel-level vulnerabilities.

Given the lack of a full operating system and the presence of just the needed libraries and dependencies, the overhead associated with a container, even though still present, is much lower than the overhead of a VM. It is, by the way, acceptable given the increased stability and reliability of the system, isolation, and manageability.

Each application running in a container is fully independent and behaves similarly to a VM; it can be started and stopped, and specific resources can be allocated to it. This allows for a shift of the architecture of the software stack from a monolithic approach, like the one that used to be used, to a system composed of a synergy of independent microservices composing the stack.

8.2.1 Docker Installation and Important Docker characteristics

For containers, specifically Docker containers, to run, it is necessary to install Docker Engine on the host machine.

Docker can be installed with just a few commands to be run on the host machine. All the commands must be run by an administrator with *Sudo* privileges. Since the machine is remote, an SSH connection was used using mRemoteNG to manage the connection.

Listing 8.1: command to uninstall all conflicting packages

```
for pkg in docker.io docker-doc docker-compose podman-docker containerd
    ↪ runc; do sudo apt-get remove $pkg; done
```

The first command, Listing 8.1, is used to remove old conflicting Docker packages (if present).

After all the old software has been removed, the Docker *apt* repository is added to the system using commands in Listing

Listing 8.2: commands to setup Docker *apt* repository

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o
    ↪ /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture)
    ↪ signed-by=/etc/apt/keyrings/docker.asc]
    ↪ https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Finally, it is possible to install the latest Docker version (Listing 8.3).

Now that Docker is installed, the server is ready to deploy and run containers using the CLI tools.

Listing 8.3: commands to install Docker

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
    ↪ docker-buildx-plugin docker-compose-plugin
```

It is possible to create custom Docker images for special applications, but Docker offers a platform called Docker Hub where it is possible to download pre-packaged images directly from the software house producing the software. Since Docker images are containers, they come pre-packaged with all the elements needed to run the application, guaranteeing the application to work properly on the Lightsail server as in the developers' computers.

Docker images are immutable templates. Containers are ephemeral, meaning that every data written inside them is maintained until the container exists, but lost once the container is removed or replaced, such as during an update. This means that security and firmware updates can not be installed from inside the container itself, since they would get lost, but they are handled at the image level directly from the image creator, guaranteeing proper software functionality and compatibility. To perform an update, a new image is generally pulled from Docker Hub. The ephemerality of the container also means that possible corruptions get reversed by updating or re-pulling the image. To re-pull means to re-download and re-run the same image. The problem is that even databases and configurations get eliminated when the container is re-created, an unacceptable trade-off. Docker Volumes and Binds are persistent storage mounts used to solve the problem. Docker volumes are storage mounts managed by Docker and physically stored on the host system. Their contents are managed by Docker and are not intended for direct manipulation on the host. Volumes can offer better performance compared to binds. Binds are host folders that are mounted inside the container in place of some of its folders. They generally have lower performance compared to Volumes and are exposed to problems, such as filesystem permissions, that might prevent the Docker container from working. They have the advantage that, since they are a host path, the files that they contain are fully accessible outside Docker, simplifying management for small deployments.

Images on Docker Hub are identified by tags that indicate the version. The *:latest* tag is used to indicate the latest stable version available for the specific Docker.

Since all Docker images are equal, and all the configurations and files are stored in the Docker volumes and binds, it is sufficient to back up these last files to fully restore the entire system.

For ease of management, almost all the persistent storage of all the containers used was set up using binds. On the host machine, a folder called *appdata* was

created in the location */home/appdata*. Inside this folder, a series of sub-folders named after the containers they refer to were set up. Inside each of these sub-folders are all the binds.

The last Docker characteristic that needs to be discussed is networking. Each container running on a single machine is connected to a virtual local network inside the host. These networks have all the characteristics of a physical computer network. They can even be totally isolated from the physical network and have no internet access. Ports of the Docker container can then be manually mapped to ports on the host machine. This is used to enable connection in entrance form the local network to the container. To enable remote connections, the host port, mapped to the Docker port, needs to be opened on the physical network's firewall, too.

A custom Docker network called *Custom-internal-network* was created, other than the default ones. Some of the containers installed need to communicate with other containers installed on the same machine. Using a custom network, it is possible to use local DNS to resolve connections to other containers using the container's name.

8.3 Portainer

The first Docker container installed on the server is Portainer.

Docker containers are managed using a series of CLI tools. Portainer offers an alternative to the CLI with a graphical user interface that simplifies long-term management.

8.3.1 Installation

Portainer is the only Docker container that will be installed using the CLI tools; the rest will be installed from its graphical user interface.

First of all, it is necessary to create a Docker volume, called *portainer_data*, to contain all the persistent data. This is done with the command shown in Listing 8.4

Listing 8.4: Command to create *portainer_data* vloume

```
sudo docker volume create portainer_data
```

Once the volume is properly created, it is possible to download and install the Portainer Docker container using the command in Listing 8.5

Listing 8.5: Command to install Portainer container

```
sudo docker run -d -p 8000:8000 -p 9443:9443 --name portainer
    ↪ --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v
    ↪ portainer_data:/data portainer/portainer-ce:lts
```

The command can be broken down into different parts:

- *sudo docker run -d*: Docker run is the command to create and run a new container. It has to be run with *sudo* privileges. *-d* is used to run the command in *detached mode*, meaning that it runs in the background.
- *-p 8000:8000*: This maps the host port 8000 to the container port 8000
- *-p 9443:9443*: This maps the host port 9443 to the container port 9443. Port 9443 is used to connect to the web user interface of Portainer.
- *- -name portainer*: This sets the name of the container created.
- *- -restart=always*: This sets the container to automatically restart if a crash may occur or if the Docker service gets restarted.

- *-v /var/run/docker.sock:/var/run/docker.sock*: This mounts a Docker bind of the host docker.sock with the container's docker.sock. This gives Portainer the capability to control Docker on the host machine.
- *-v portainer_data:/data*: This mounts the containers'*/data* directory to the Docker Volume *portainer_data* created in Listing 8.4. This allows the files in */data* to be preserved.
- *portainer/portainer-ce:lts*: This specifies the Docker Hub'image to be downloaded. *lts* is the specific tag of the Long Term Support version

8.3.2 Features

If proper port forwarding is set up on the firewall, once the Docker container is downloaded, properly installed, and started, it is possible to connect to https://<server-IP-adress>:9443 and access the Portainer graphical user interface from every computer connected to the internet. *3.76.128.16* is the public IPv4 address of the Lightsail server. At first boot, it asks to set up the administrator user and password. Afterwards, it asks to select the environment that the user wants to connect to. Figure 8.4. Portainer is an extremely powerful tool capable of managing entire clusters of multiple servers from a single user interface. Since there is just one server configured, it is necessary to press on *Local* (Figure 8.4)

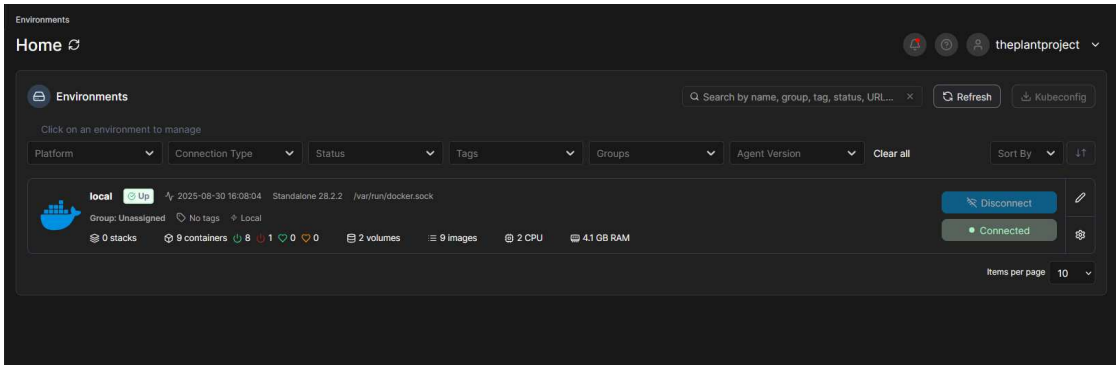


Figure 8.4: Portainer environment selection

Once the local environment is set, the dashboard opens up, Figure 8.5. From here, it is possible to open the page to set up the Docker networks, manage and create Docker Volumes, manage the Docker images downloaded, and, most importantly, manage the deployed Docker containers. (Figure 8.5)

Once the container management page is opened, a list of all the installed containers, both from the Portainer interface and from the CLI, is presented to the user. Figure 8.6 shows how the page presents once the entire IoT container stack for this thesis was deployed, plus some other containers added to extend the monitoring capabilities to other projects of the research group.

Each line refers to a different container. It indicates the name, the state, some quick actions on the container, the image with the tag, the date of creation, the IP address on the Docker network, the ports that have been bonded from the host to the Docker, and the ownership of the Docker, important in case multiple users are using the same instance of Portainer.

Figure 8.7 focuses on the Quick Actions buttons of the "containers" page of Portainer. Starting from the left, the first button opens the Logs of the containers, an essential tool during debugging. The second button, "Inspect", opens a list of all the configurations of the Docker container. The third, "Stats", is used to monitor

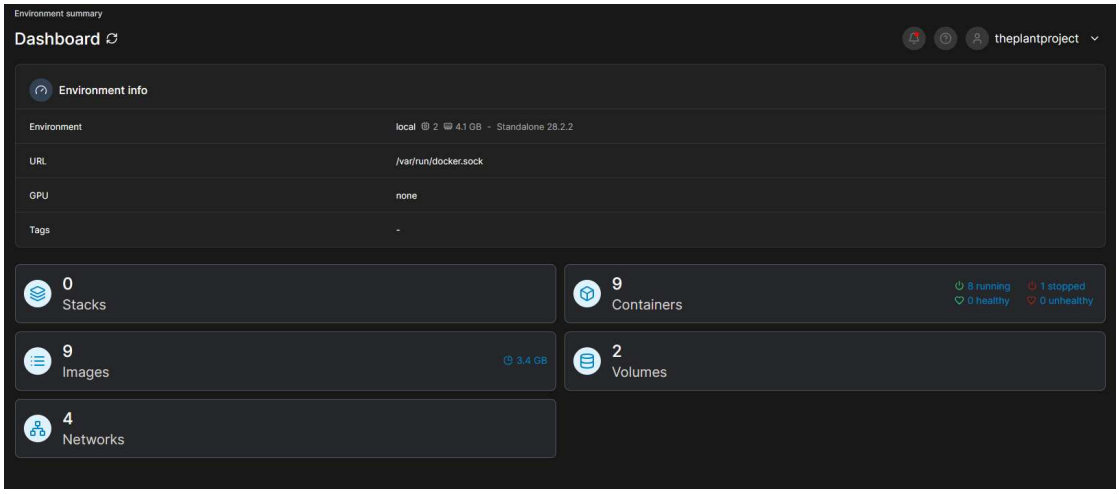


Figure 8.5: Portainer dashboard

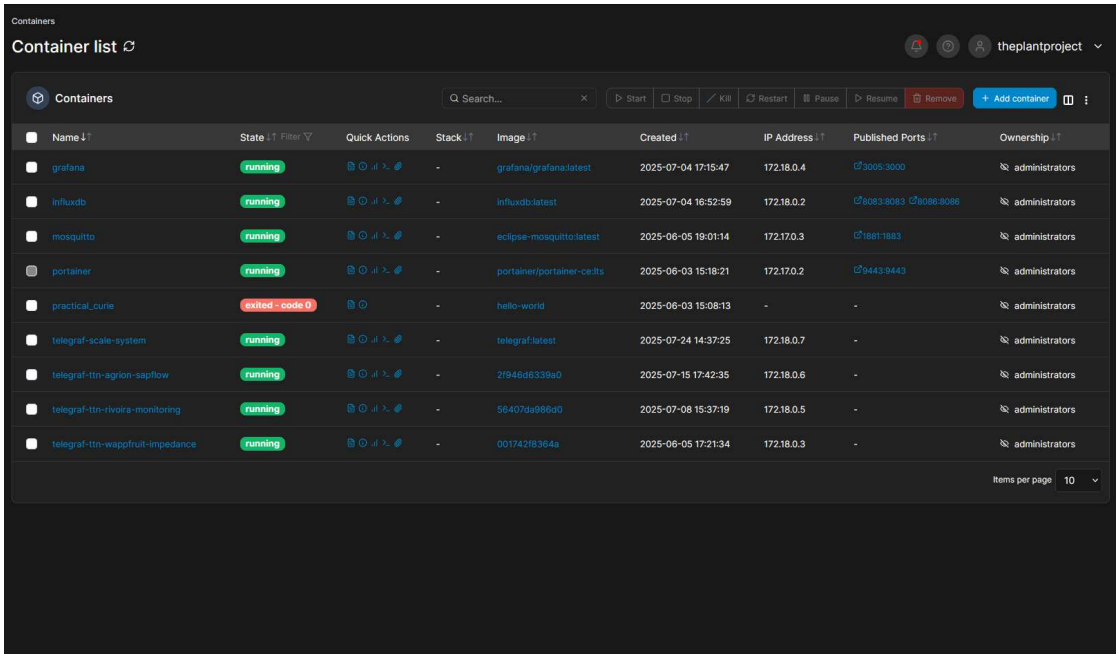


Figure 8.6: Portainer container page

the resource usage. The last two buttons are used to connect a terminal to the Docker container. Even though not actually completely true, each Docker behaves like an independent machine, and connecting a terminal is similar to establishing an SSH connection to a remote server.

Going back to the container list in the "containers" page, it is possible to select

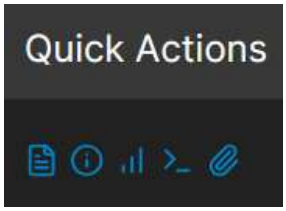


Figure 8.7: Detail on Quick Actions from the "container" page

containers by pressing on the white square on the left of each line. Once a container is selected, it is possible to control its state using the buttons on the top right corner of the list and reported in Figure 8.8.

Figure 8.8: State control buttons

The "start" button is used to turn on a powered off docker, the "stop" button is used to gracefully turn off a docker, while "kill" is used to forcefully turn it off. "restart" reboots the Docker. "pause" and "resume" are used to momentarily freeze and unfreeze the execution of a specific docker without shutting it down. Finally, "remove" eliminates the Docker container. The container gets removed, but the files in the Volumes and in the binds do not get touched. This means that a Docker container can be re-created at any time exactly as it was when it was eliminated.

The last button is not related to the state of the container. "+ Add container" is used to create a new container; its usage will be discussed in detail in Section 8.4.

Clicking on the container name opens a page with all the container details. Now, to the top bar of the Figure 8.8, two buttons get added, and the "+ Add container" gets removed. The first one is the "recreate" button that restarts the container from the image. All the non-persistent data not saved in a bind or a volume will be lost. There is also the possibility to re-pull the image; this means that the image will be re-downloaded from Docker Hub, substituting the local one. The second button is "duplicate/edit". This option behaves similarly to the "+ Add container", but all fields are pre-filled with the configuration of the specific Docker container selected. If the container name gets changed, the Docker container gets duplicated, while if the name is not changed and other fields get changed, the Docker container updates its configuration to the newly configured one.

Other than the updated top bar, the page contains a summary of all the configurations of the Docker container.

8.4 IoT Stack

The Lightsail VPS with Docker, managed through Portainer, creates the solid foundation on which the entire IoT stack is built. The analysis of the containers used and their installation will be presented following the sequence of data flow, beginning from the container that first receives the sensor data aggregator's readings and ending with the container that presents the final data to the user. The flow is schematized in Figure 8.1 and briefly introduced in section 2.4.

8.4.1 Eclipse Mosquitto

Figure 8.9: Eclipse Mosquitto logo

Eclipse Mosquitto is an open source MQTT broker. Its function, as explained in section 2.4, is to coordinate the MQTT transmissions. Messages are transmitted to and from the Raspberry Pi controlling the scales using exclusively MQTT.

Installation

The installation procedure starts by pressing the "+ Add container" shown in Figure 8.8. Once pressed, a page divided into two subsections appears.

The top subsection, shown in Figure 8.10, contains the more general settings for the Docker container. Of particular relevance here is the name, used to identify the container. If two containers have identical configurations but different names, the two are considered distinct containers. On the other hand, it is impossible to have two containers with different configurations sharing the same name since the second container created would overwrite the first one.

The "+ Add container" page is a graphical user interface to generate and store a *docker run* command similar to the one shown in listing 8.5.

The section called "Image Configuration" is used to set up the registry from where the image gets downloaded, like Docker Hub, and the specific image name with the specified tag.

In the "Network ports configuration" section, it is possible to set up port mappings.

Finally, in the bottom left corner of the top subsection is present the button used to deploy the container once configured.

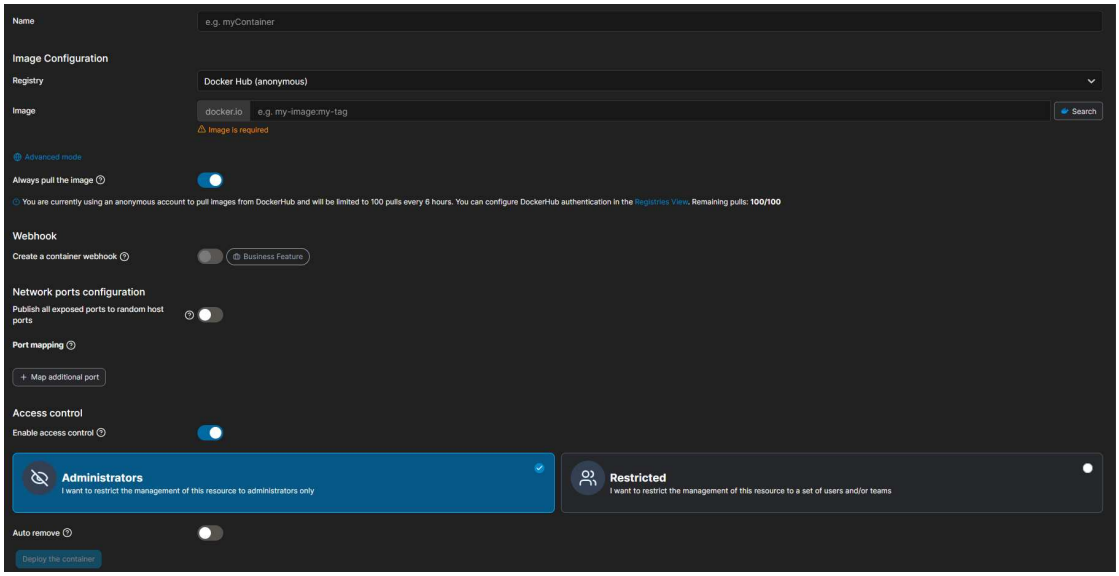


Figure 8.10: Top subsection in new Docker container creation page

The Eclipse Mosquitto container was called "mosquitto". The image is downloaded from Docker Hub, and it is the official image produced by Eclipse, available under the name "eclipse-mosquitto:latest". The last part of the image name, ":latest", is the tag that is used to indicate the version that needs to be downloaded. ":latest" indicates the newest stable release.

The standard port for MQTT is port 1883, and Eclipse Mosquitto expects connections on this port, but it was decided to keep this specific port free on the server for further use. Using port mapping, the 1883 port of the container was mapped to port 1881 of the server. See Figure 8.11.

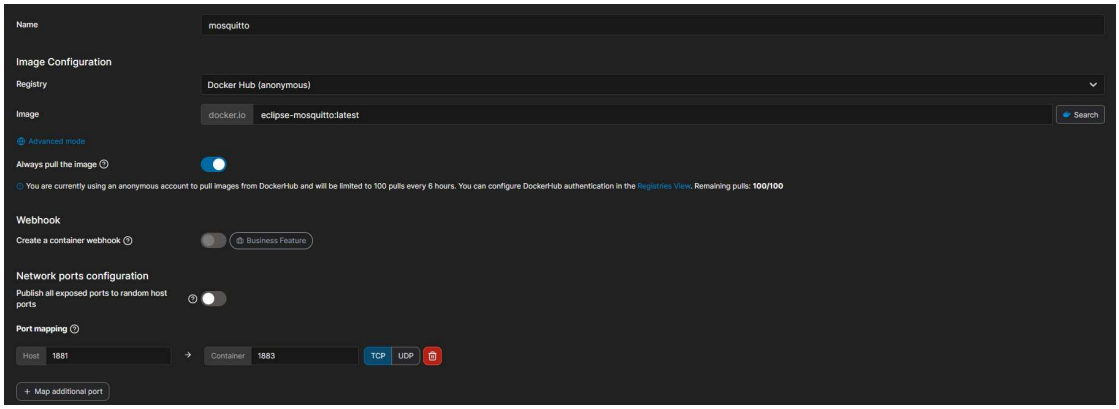


Figure 8.11: Top sub-section of Mosquitto configuration

The lower subsection of the page contains the advanced container settings. It is organized into eight sub-pages, of which the most relevant ones are shown in Figure from 8.12 to 8.18.

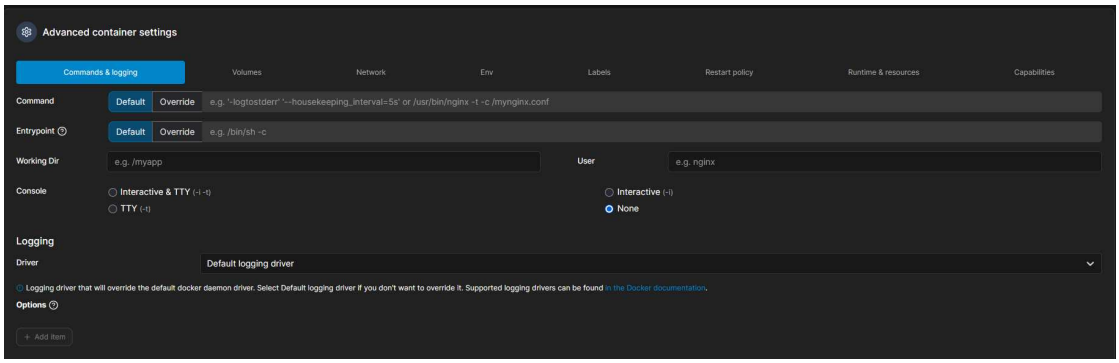


Figure 8.12: Commands and logging settings

The first of these pages is "Commands & logging". It is used to set extra commands during *docker pull*, specifying the type of console to use and the logging driver.

For Mosquitto, all these settings were left to their default configuration.

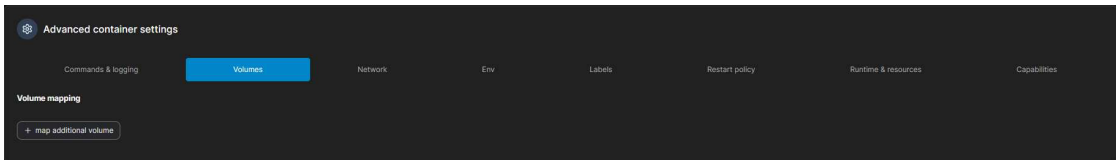


Figure 8.13: Volumes and Bindings

The second page, "Volumes," is used to set persistent storage such as Docker Volumes or Binds. Pressing the "+ map additional volume" button, it is possible to define the container bath to map and, depending on whether the volume or bind option is selected, the name of the path on the host or the Docker volume created beforehand in the "Volumes" page of Portainer present in the dashboard (see Figure 8.5).

Mosquitto needs three directories to be persistent. These are */mosquitto/config*, */mosquitto/data* and */mosquitto/log*. Figure 8.14 showed the relevant configuration portion of the "Volumes" section.

As explained in subsection 8.2.1, it was decided to use almost exclusively binds to set up the persistent storage. In the host directory */home/appdata*, a folder for each Docker container was created, and in these folders, all the bindings for that container are placed. The containers' folders are respectively

Figure 8.14: Mosquitto binds

mapped to */home/appdata/mosquitto/config*, */home/appdata/mosquitto/data*, and */home/appdata/mosquitto/log*.

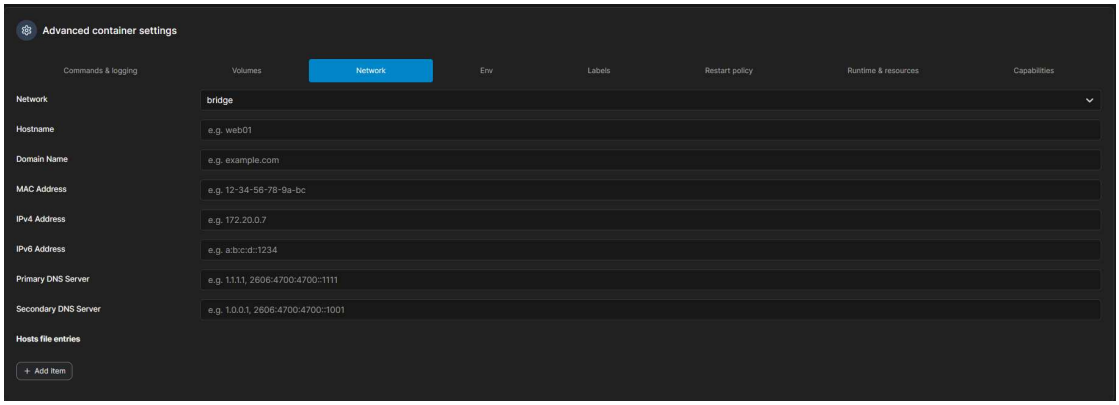


Figure 8.15: Network settings

The "Network" page is used to set all the network-related parameters of the Docker, such as IP addresses, DNS server, and, most importantly, the Network the container is connected to. Using the drop-down menu, it is possible to select:

- bridge: This is the default option. The container is connected to a virtual local Docker network, which is then bridged to the fiscal network the host is connected to. When a container is connected to the bridged network, it gets assigned, if no static address is set up, a local IP address inside the Docker network. All the ports that need to be exposed need to be manually mapped. The container is isolated from the host network.
- none: No network is assigned. The container is fully isolated.

- host: The container is connected to the host network, sharing its IP address. Since this is not an isolated network, it is not necessary to perform port mapping, since when a container opens a port, it is automatically opened also on the server. This setting might lead to conflicts.
- custom Docker network (such as *Custom-internal-network*): the user can define additional Docker networks with specific characteristics to suit their needs. *Custom-internal-network* was created similar to Bridge, but with the added characteristics of having a local DNS, giving the ability to containers to connect using their container name instead of their IP address.

Mosquitto does not require any particular network setting, and everything on this page was left to its default configuration.

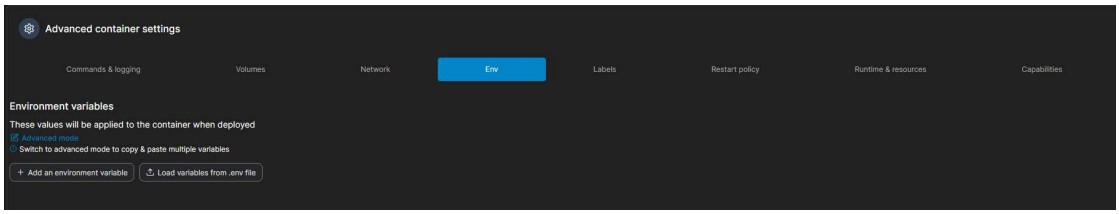


Figure 8.16: Environment variables

Environment variables are key-value pairs used to configure behaviours without directly modifying their code. They are commonly used, for example, to specify the time zone, external services API keys, and others. Environment variables are set using the "Env" tab shown in Figure 8.16.

Mosquitto does not require any particular environment variable to be configured.

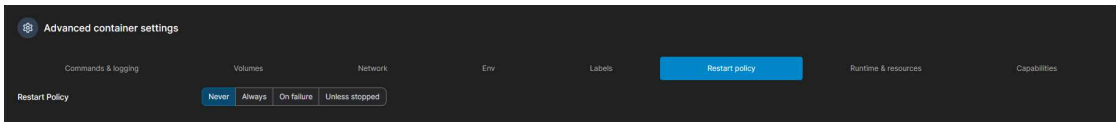


Figure 8.17: Restart policy options

The "Restart policy" page is used to configure how the container should behave in case of exit due to a system reboot or even a crash. Figure 8.17 shows the available options:

- Never: the default one. The container does not get restarted
- Always: the container is always restarted
- On failure: the container is restarted only if the exit was caused by a failure condition

- Unless Stopped: the container is always restarted as long as the exit was not manually triggered by the user.

All the containers deployed, including Mosquitto, are set to *unless stopped*.

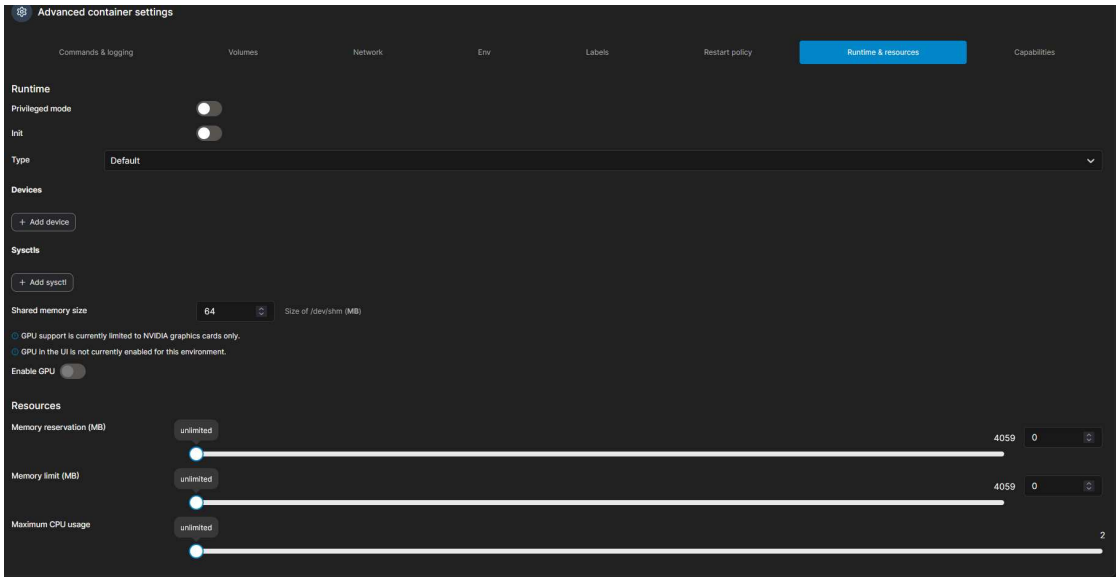


Figure 8.18: Runtime and resource constraints

The last page discussed is "Runtime & resources". This page, Figure 8.18, is used to set the hardware device (such as USB devices, graphics cards,..) that can be attached to a container.

Additionally, this page can also be used to set the resources available to a single Docker. By default the Docker is allowed to use as many resources as needed.

Mosquitto uses the default configuration.

Configuration

Once installed, the Mosquito instance needs a *mosquitto.conf* file in the container location */mosquitto/config*. This is a text file that defines the behaviour of the application. If this file is not present, the container won't boot. The mosquitto.conf file used is shown in Listing 8.6

```
allow_anonymous false
listener 1883
persistence true
password_file /mosquitto/config/pwfile
persistence_file mosquitto.db
persistence_location /mosquitto/data/
```

Listing 8.6: mosquitto.conf file

The first line of Listing 8.6 is used to disable anonymous connections. Anonymous connections allow anyone who knows the IP address and port of the broker to connect and use it. This could be a somewhat desirable feature when the broker is only exposed to a secured local network, since it simplifies the connection of nodes, but it is not acceptable in a broker exposed to the internet.

The second line sets the port used by the broker for connections. Port 1883 is the default MQTT port. This port will later be mapped to a different port on the Host system.

Line three enables persistence mode, which allows the gateway to maintain data, such as subscriptions, queued messages, and others, when the server gets restarted. Lines five and six set the name of the database used for persistence and its location.

Finally, line four sets the position for the password file, a file containing usernames and passwords of the users. Usernames are essential since anonymous connections are disabled.

Other than the *mosquitto.conf*, it is necessary to create at least one account to give the possibility for sensors to connect to the broker.

Listing 8.7: New user creation comand

```
mosquitto_passwd -c ~/mosquitto/config/passwd ThePlantProject
```

Listing 8.7 shows the command used to create a new user. This command has to be executed from a terminal directly connected to the container. It is accessible by pressing the second button from the right in the "Quick Actions" panel, shown in Figure 8.7. Once pressed, it opens the page shown in Figure 8.19.

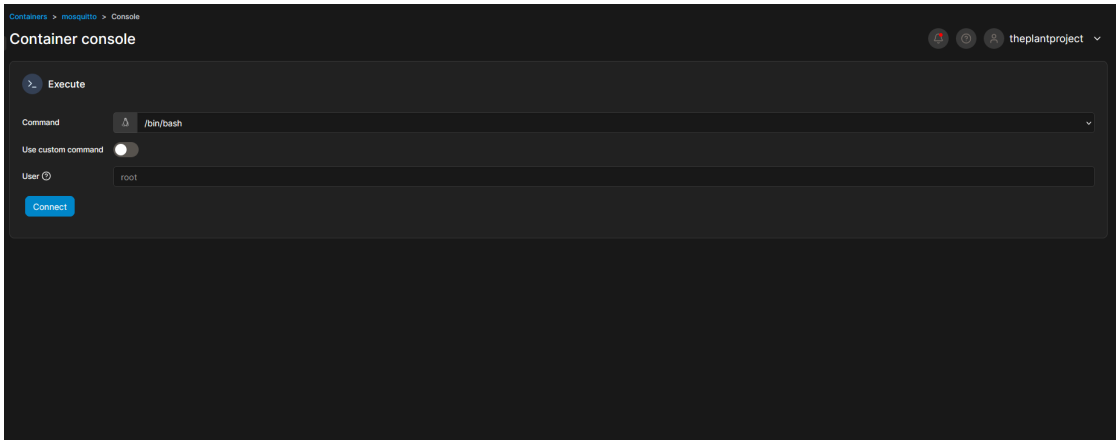


Figure 8.19: Container console page

Once on this page, it is essential, for Mosquitto, to change using the drop-down menu the command from */bin/bash* to */bin/sh*. Once done, it is possible to click the "Connect" button, and a terminal in the Docker container should open.

Command shown in Listing 8.7 can now be executed. *mosquitto_passwd* is the tool used to create new users, *-c* is used to create the password file if it does not yet exist in the location set by *~/mosquitto/config/passwd*. Finally, *ThePlantProject* is the username of the new user.

Once the command is executed, the user is prompted to enter a password for the just-created username.

Mosquitto is now configured and working.

8.4.2 InfluxData Telegraf

Figure 8.20: InfluxData Telegraf logo

The data received by Elipse Mosquitto is, generally speaking, volatile. If no device is ready to receive it once it is transmitted, if operating with a QoS of 0, this data is lost. It is necessary to have a database capable of long-term storage of the collected data. The database, on its own, is not capable of directly receiving MQTT messages. Telegraf is the tool used to receive the data using MQTT, decode the JSON file transmitted, prepare the data, and send it to the database to be permanently stored.

Installation

The installation process through Portainer is similar to what was explained in subsection 8.4.1. The container was called "telegraf-scale-system". This choice, rather than naming it simply "telegraf", was made because, as shown in Figure 8.6, the system has already been extended to various monitoring needs of the research group. For ease of management, at the planning stage, it was decided to deploy a separate Telegraf instance for each separate monitoring application. Each container, therefore, requires an explicit name that reflects its purpose.

The image is downloaded from Docker Hub directly from the official InfluxData repository. The image name is *telegraf:latest*

There is no need to configure any port mapping since no open ports are required.

Going to the Advanced container setting, the "Commands & logging" as well as the "Env", "Labels", "Runtime & resources", and "Capabilities" are left to their default configuration.

As shown in Figure 8.21, it is necessary to map container' */etc/telegraf* directory to host' */home/appdata/telegraf-scale-system* directory to guarantee persistent storage for settings.

In the networking setting page, the network was set to the custom network called *Custom-internal-network*. (Figure 8.22)

The "Restart policy" was set to "Unless stopped"

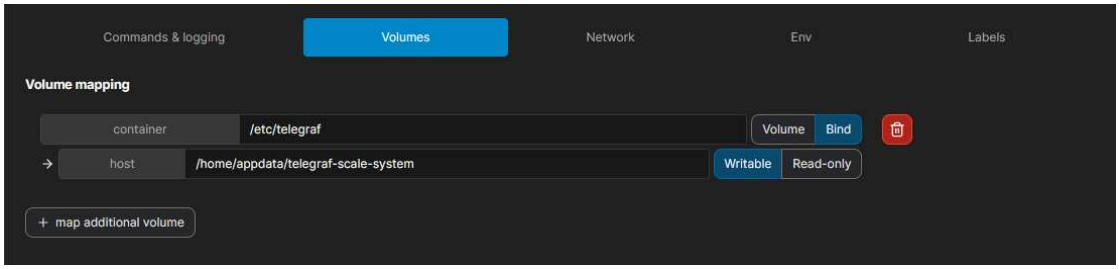


Figure 8.21: Volumes configuration page

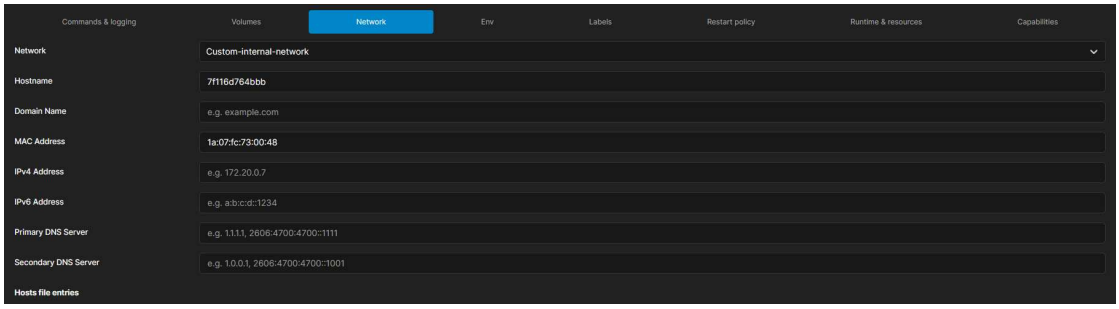


Figure 8.22: Network configuration page

Configuration

Similar to Mosquitto, Telegraf requires a configuration file to work. This file has to be called *telegraf.conf* and placed in the persistent directory.

```
#########################
# Telegraf Configuration
#########################

[agent]

  debug = true
  quiet = false

  hostname = "telegraf-scale-system"

#########################
# Input
#########################

[[inputs.mqtt_consumer]]
  servers = ["mqtt://3.76.128.16:1881"]
  username = "ThePlantProject"
  password = "**********"

  topics = ["scales/+/sensors/average"]

  client_id = "telegraf-scale-system"
  persistent_session = false

  data_format = "json_v2"

  [[inputs.mqtt_consumer.topic_parsing]]
    topic = "scales/+/sensors/average"
    measurement = "_/measurement/_/_"

  [[inputs.mqtt_consumer.json_v2]]
    timestamp_path = "timestamp"
    timestamp_format = "RFC3339"

    [[inputs.mqtt_consumer.json_v2.field]]
      path = "average_0"
      type = "float"
      optional = true

    [[inputs.mqtt_consumer.json_v2.field]]
      path = "average_1"
      type = "float"
```

```
44        optional = true
45
46      [[inputs.mqtt_consumer.json_v2.field]]
47        path = "average_2"
48        type = "float"
49        optional = true
50
51      [[inputs.mqtt_consumer.json_v2.field]]
52        path = "average_3"
53        type = "float"
54        optional = true
55
56  ########################
57  # Output
58  ########################
59
60  [[outputs.influxdb_v2]]
61    urls         = ["http://influxdb:8086"]
62    token        = "***************************"
63    organization = "polito"
64    bucket       = "test_scale"
```

Listing 8.8: telegraf.conf file

```
1  {
2  "timestamp": "2025-09-02T12:04:38.803181+00:00",
3  "average_0": 4.527593533566255,
4  "average_1": 3.423533563462578
5  }
```

Listing 8.9: Example of a JSON used to transmit the scales readings

Listing 8.8 is the *telegraf.conf* used to set up Telegraf, while Listing 8.9 is an example of the JSON payload received containing the readings of the scales.

The configuration file is divided into three subsections. It starts with the Agent configuration, continues with the Input configuration, and ends with the Output configuration.

Agent configuration affects the overall collection and batching behaviour of Telegraf. "Debug", line 7 of Listing 8.8, increases the logging verbosity, while the disabled "quiet" mode allows printing of regular log messages. Finally, the "hostname" is used to tag the sent measurements with the Telegraf instance name.

The second section is about the input of data. Telegraf to perform its functions is based on a series of input and output plug-ins used to move data from one to the other. The input plug-in used is"[[inputs.mqtt_consumer]]" used to receive data through MQTT.

To receive MQTT messages, Telegraf needs to subscribe to the same broker where the messages are sent. Lines seventeen, eighteen, and nineteen (of Listing 8.8) are used to subscribe to the Mosquitto instance configured in subsection 8.4.1. Telegraf has to be subscribed to the correct topic where the relevant messages are sent. This is done in line twenty-one. The topic is *scales/+/sensors/average*. The "+" sign is a wildcard. Wildcards are used to replace every single word, meaning that Telegraf will subscribe to all the topics that start with *scales/* and finish with */sensors/average* without caring about the exact subtopic substituted by the "+" sign. This is done because the Python code running on the Raspberry Pi will use this exact field to tag its measurements, indicating the node that created them. Please refer to subsection 7.3.7 for more details. Using the wildcard, Telegraf will receive the messages from all the sensor data aggregators, no matter their number.

"Client_id" is used to identify the client when connecting to MQTT.

"persistent_session" is used to keep subscription state when disconnecting. It is disabled because it was experimentally verified that, in some instances, it caused disconnection problems from the broker.

"Data_format" specifies what kind of payload is received through MQTT.

Lines twenty-eight to thirty are used to extract the name of the sensor node transmitting the data from the topic of the received message. In the definition of the topic subscription it was used the "+" wildcard was used to ignore the part of the topic referring to the sensor node producing the information. Now, just this part of the topic is used to set the measurement name.

Finally, the data is extracted from the payload.

Listing 8.9 shows a typical JSON payload. The first element present is the timestamp identifying when the measurement was produced. After the timestamp are all the readings identified by "average_0", "average_1", "average_2", "average_3", depending on the channel producing the reading. Listing 8.9 does not have the full set of readings present; this means that a scale is connected just to the first two channels, and the rest of the channel readings are not sent by the edge Python code.

The input plug-in starts by extracting the timestamp, specifying it is in *RFC3339* format.

After the timestamp is extracted, from line thirty-six to fifty-four, four almost identical sections of code extract the reading from the various channels. The reading type is specified to be a floating point, and it is essential that "optional" is set to true. This allows the system to ignore the fact that there are going to be missing channels in the payload when not all the scales are populated. If "optional" is not correctly set, no data will be sent to the database since Telegraf will stop in an error state.

The final part of the code is the connections to the database. Data is sent using the APIs through the network. It is necessary to set up a token to authenticate to

the database. More details on the database, its organization, and how the token is generated are available in subsection 8.4.3.

In line 61, the internal DNS server available through the custom Docker network is leveraged, which allows the connection to the InfluxDB container just using its name.

8.4.3 InfluxData InfluxDB

InfluxDB is the selected database. A database is used to long-term store the readings and make them available to users. InfluxDB is a database specifically optimized for time series, meaning it is built to most efficiently store data points that are associated with timestamps.

Data in InfluxDB is organized into *Buckets*. Each bucket is associated with a specific retention policy that allows for the deletion of data after a certain time interval has passed. For this project, the bucket was set to never delete the data. Inside each *Bucket* are *Measurements* that contain *Tags*, *Fields*, and *Timestamps*. *Fields* are the actual readings, *Tags* are values that are stored as strings and metadata, and mainly used for filtering and grouping. Using an analogy with a more standard database type, *Measurements* are the tables. Tables are built of columns, the *Fields* and *Tags*.

For this project, a *Bucket* called *test_scale* has been created. Data is grouped using Telegraf in *Measurements* called as the Raspberry Pi, sending the data (for example: pimaster1, pimaster2, ...). The *Fields* are the readings from the scales, so average_0, average_1, and so on. There are also two tags: the first one is the host that sends the data to the database, in this case it is *telegraf-scale-system*, and the second one is the MQTT topic from which the message was received, for example *scales/pimaster1/sensors/average*. (see Figure 8.23)

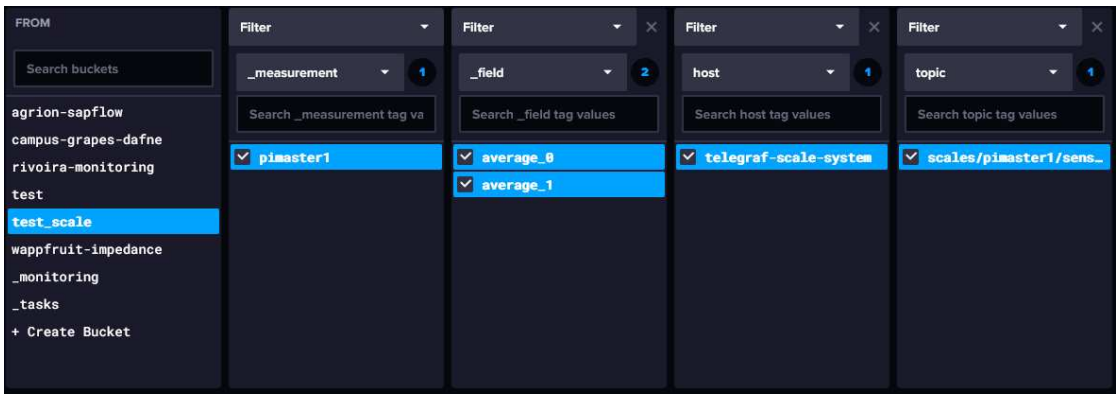


Figure 8.23: InfluxDB data structure

Installation

The installation process is similar to the containers already discussed.

The container is named "influxdb" and the image, "influxdb:latest", is obtained from Docker Hub.

It is necessary to map port 8083, used for the web interface. Since there are no conflicts, it was decided to map port 8083 on the container to port 8083 of the host machine.

As for what it is concerning the "advanced container settings", only "Volumes", "Network", and "Restart policy" have been modified.

The container's path */var/lib/influxdb2* was mapped to */home/appdata/influxdb/db2* on the host, path */var/lib/influxdb* on the container to */home/appdata/influxdb*, and a Docker Volume for the container's path */etc/influxdb2*.

In the network tab, the container was connected to "Custom-internal-network" to use the internal DNS feature.

The "Restart policy" was set to "unless stopped".

Configuration

Once started for the first time, the user is prompted to create an administrator account, to create an organization (that is, a workspace for a group of users), and to create the first Bucket.

The organization created is "polito" while the bucket created to store the values coming from the scales is "test_scale".

It is necessary to create an API token that allows Telegraf to load data to the database.

Figure 8.24 is used to illustrate the procedure to create a token. First of all, the menu on the left is used to navigate to the "Load Data" section, and then the "API token" subsection.

In this page, a button in the top right corner is used to create the Token.

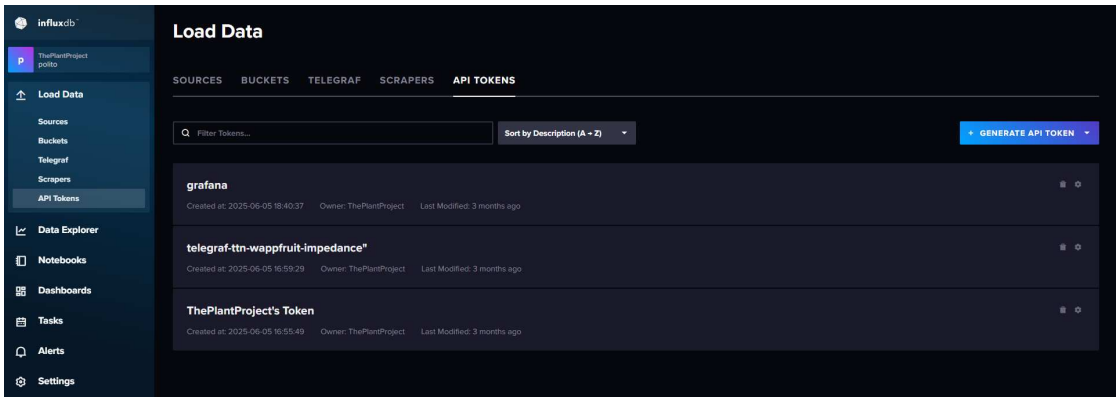


Figure 8.24: API token page of InfluxDB

The just-created token, together with the organization name and the bucket, is used in the telegraf.conf file, Listing 8.8, to authenticate Telegraf to the InfluxDB server.

InfluxDB is a schema-less database. This means that the database structure is not defined at the design stage, but it is defined directly by the incoming data. For this reason, no extra configurations are needed on the database.

8.4.4 Grafana

The scale data has now reached its final stage: it is safely stored in the database and ready for use. For formal analysis, the InfluxDB APIs can be accessed directly from Python or MATLAB to retrieve the data. However, for a quicker, high-level visualization to understand the measurements at a glance, Grafana is used.

Figure 8.25: Logo Grafana

Grafana is a platform for visualization and monitoring. It connects to databases, such as InfluxDB, and it is used to create dashboards with graphs and charts, and to set alerts based on the data stored in the database.

Installation

The Docker container is called "grafana". The image is downloaded from Docker Hub and called "grafana/grafana:latest". In the basic configuration, it is assigned a port mapping that assigns the container port 3000 to the host port 3005, used for the web GUI.

In the Advanced settings, a bind of the container folder */var/lib/grafana* to host */home/appdata/grafana* was configured. The networking was set to use the custom Docker network "Custom-internal-network".

Except for the "restart policy" set to "unless stopped", no other settings were manually modified.

Configuration

Grafana is built to visualize data. The first thing that is necessary to do is to connect it to a data source, InfluxDB in this case, containing the data of interest.

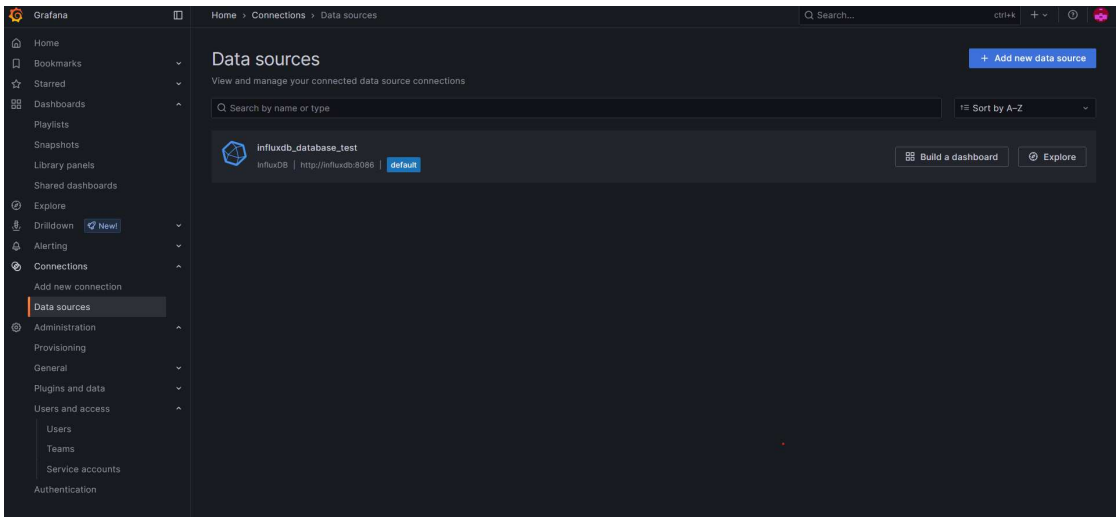


Figure 8.26: Grafana data sources page

To reach the data sources configuration page, once entered in the web GUI, it is necessary to press on "Connections" and then "data sources" using the menu on the left. The page shown in Figure 8.26 opens. Using the blue button in the top right corner, it is possible to add a data source.

Once pressed, the user must select InfluxDB. Clicking opens a configuration page. (Figure 8.27)

The first thing to configure is the name of the connection, in this case "influxdb_database_test".

A query language is a specialized programming language used to make requests to a database. Flux is a query language created by InfluxData to be used with InfluxDB.

The "URL" should contain the IP address and the correct port of the InfluxDB instance. Since a custom Docker network is used, it is possible to use the container name instead of the IP address, as already done in the telegraf.conf (Listing 8.8)

In the "InfluxDB Details" section, the Organization ("polito") is configured to be used, and an API token of InfluxDB is added, generated as shown in subsection 8.4.3.

Once all these fields are filled, the configuration can be saved and exited.

In chapter 9 will be provided many different examples of visualization created in Grafana. Visualizations are interactive graphical representations of data.

To create a visualization, it is first necessary to create a dashboard.

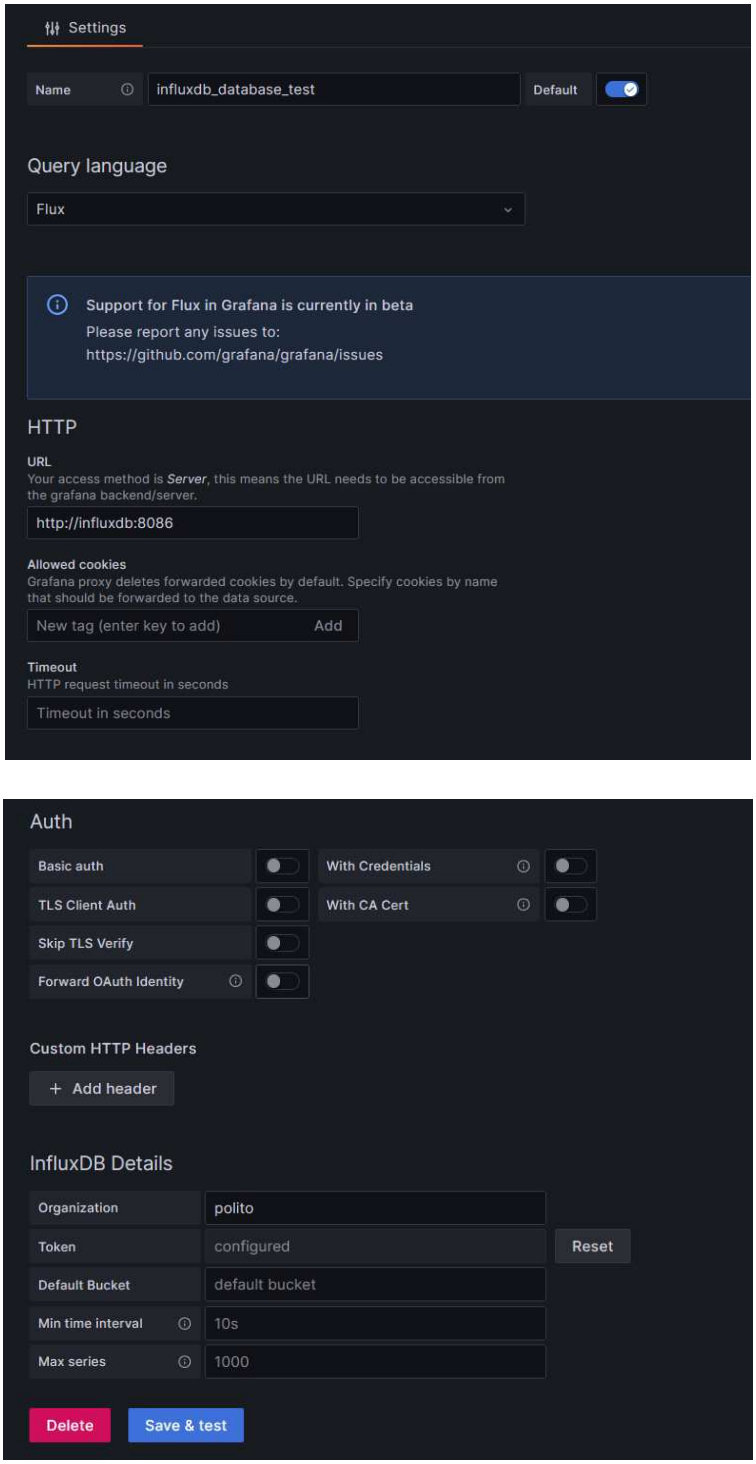


Figure 8.27: Grafana data source configuration

To create a dashboard, it is first necessary to use the menu on the left and navigate to the "Dashboard" page. In this page, the "New" blue button in the top right corner is used to create a new dashboard.

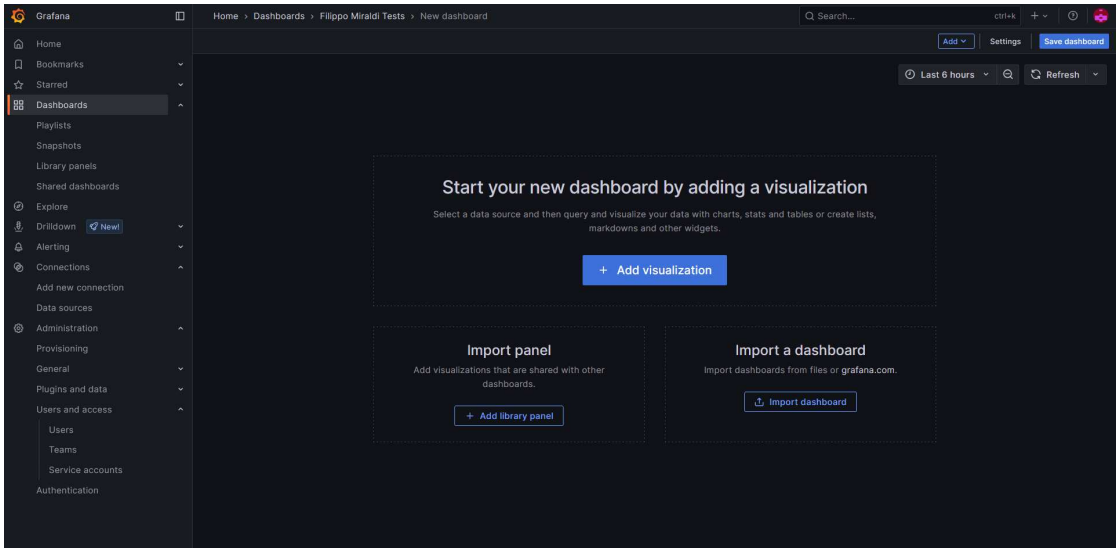


Figure 8.28: Grafana new dashboard page

A new empty dashboard is generated, Figure 8.28. The top right "Add" button is used to add a visualization.

Once pressed, the user is prompted to select the data source; in this example will be used the just-created *influxdb_database_test* data source.

Once the data source is selected, the page shown in the example of Figure 8.29 is automatically opened.

This page is divided into three subsections. On the top left is the preview of the visualization. On the bottom left is the database query corresponding to the data that needs to be visualized. The right column contains the menu that allows you to select what kind of visualization is wished and to personalize it. All sorts of visualizations are available, from time series to histograms and heatmaps.

To query the data of the database Flux language is used. Once the user sets the time range and refreshes the dashboard, using the dedicated buttons on the top right corner, the data is queried from the database on the fly and immediately displayed to the user.

Grafana has the added capability of calculating statistics on the data (such as standard deviation, mean, median, ...) and to calculate transformations on the data to further boost the quality of its visualization.

The capabilities of Grafana visualizations are vast and cannot be properly covered in just a couple of paragraphs. An extensive documentation is available on

the Grafana website containing details of all its extensive capabilities. [32]

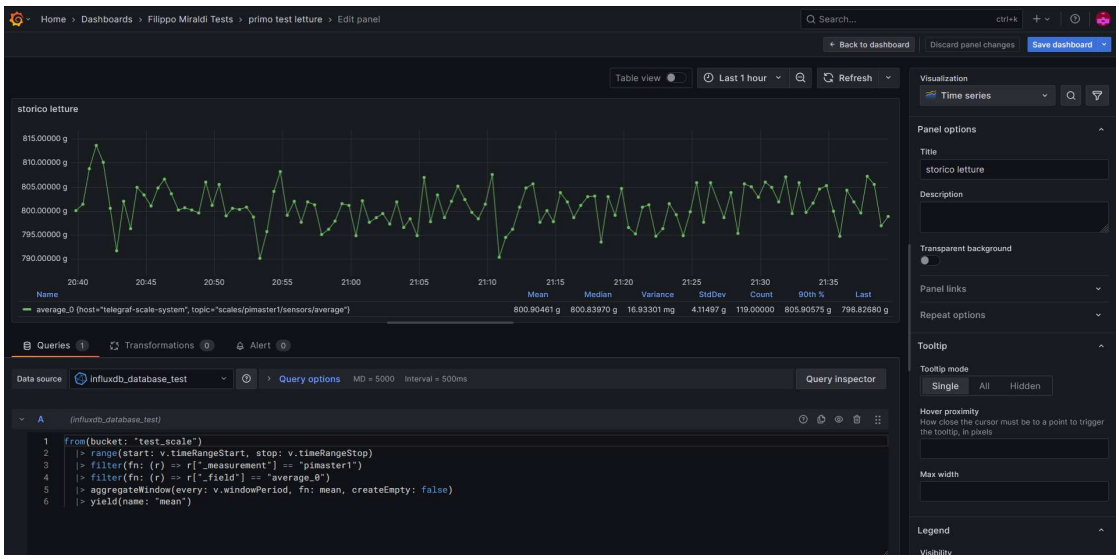


Figure 8.29: Grafana visualization creation page

Chapter 9

Performance evaluation and validation

Disclaimer: Due to logistical reasons, the performance evaluation and validation were conducted using an ***L6N-C3-50kg*** load cell with a capacity of 50kg **instead of** the ***L6N-C3-10kg*** load cell with a rated load of 10kg that was selected for the final version of the project. The five times greater rated capacity of the temporary load cell translates into a five times smaller signal for the same applied load. Ideally, **the usage of the final 10kg load cell is expected to produce five times better results** than what is presented in this chapter.

9.1 Experimental Setup

A Raspberry Pi was prepared following the instructions provided in Appendix C. The SBC was fitted with a Pi_master shield with a Slave_zssc3224 connected to channel 1 and Slave_nau7802 connected to channel 0. A *CELMI LP7611* scale fitted with a *L6N-C3-50kg* load cell was first connected to Slave_nau7802, leaving the inputs of Slave_zssc3224 floating. Once all the tests were performed, the scale was connected to Slave_zssc3224, leaving the inputs of Slave_nau7802 floating.

The computer was connected to the internet using an Ethernet cable. The power supply used is the official Raspberry Pi 4 power supply.

Figure 9.1 provides a block diagram of the electronic setup used.

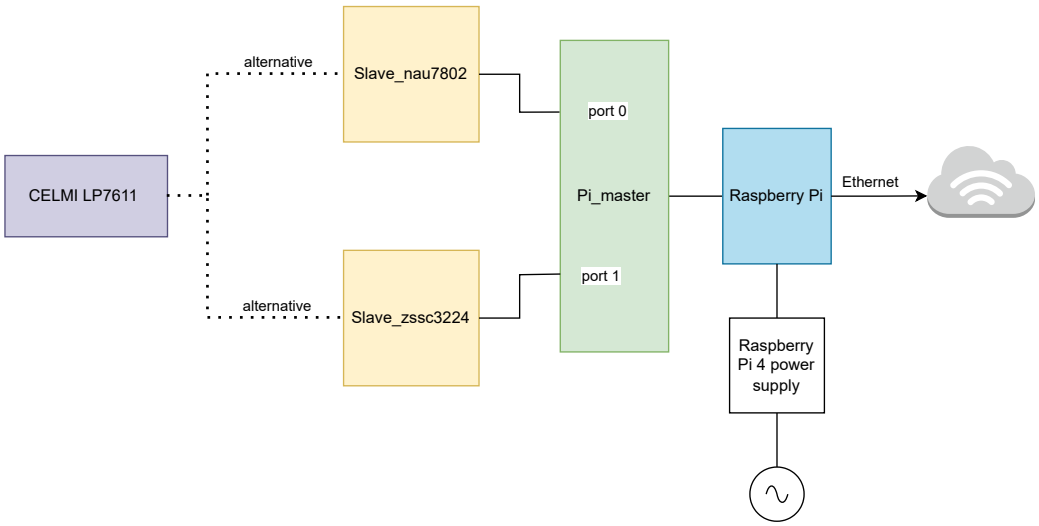


Figure 9.1: Performance evaluation and validation test setup block diagram

The scale and the electronics during the test were placed on a heavy, ≈250kg, steel workbench, ensuring good immunity to vibrations, small movements, and accidental external influences.

9.2 Experimental Procedure

The procedure is divided into a first step of calibration and later the measurement.

The scale system is left powered on for around one hour before the beginning of the calibration procedure. This is done to allow for thermal stabilization and settling of all components, ensuring consistent and accurate measurements.

It is essential to keep in mind that the voltages involved are small, and a high level of precision is required from the analog front-ends. It is crucial to consider all factors that might influence the readings to obtain good results. Table 9.1 provides some examples of the input differential voltage generated by the load cell for a 0.8 kg load and the variation of such voltage for a 1g weight change.

Parameter	**NAU7802 (4.5V bridge supply)**	**ZSSC3224 (1.68V bridge supply)**
Input with 0.8 kg load	144 μV	53.76 μV
Input change per unit load (dV/dg)	180 nV	67.2 nV

Table 9.1: Analog front-ends theoretical input characteristics with the 50Kg load cell using their typical excitation voltages.

Calibration and validation of the readings were performed using a set of precision calibration weights, shown in Figure 9.2, composed of:

- 1 x 10g weight
- 2 x 20g weights
- 1 x 50g weight
- 2 x 100g weights
- 1 x 200g weight
- 1 x 500g weight

Using the procedure described in subsection 7.3.2, the first step requires leaving the scale plate empty to correct the offset of measurement. Once the offset is corrected, a specific load of 1kg obtained by combining all the calibration weights is placed on the scale. This reading, combined with the zero measurement, is used to correct the linear response slope of the load cell.

Once the calibration is finished, the 200 g weight was removed, leaving an equivalent 800 g weight on the scale plate. 2750 measurements were then taken, representing around 22 hours of continuous sampling, first using Slave_nau7802 front

Figure 9.2: Calibration weights set

end and then the same load cell was disconnected and connected to Slave_zssc3224 front end.

The Python edge stack explained in chapter 7 and the cloud stack explained in chapter 8 were used to collect the readings and analyze them.

9.3 Results

The first analog front-end tested is Slave_nau7802.

Figure 9.3 shows a histogram of the 2750 measurements. Using Grafana's built-in tools, it is possible to calculate that the mean of all measurements is 800.86070 g, just 0.8860 g above the expected value, with a standard deviation of 4.32689 g.

The histogram is a symmetrical bell-shaped distribution, suggesting a normal distribution of the values. The symmetry is further evident in the low skewness of 0.057556. This symmetry suggests the error to be of a random nature rather than systematic.

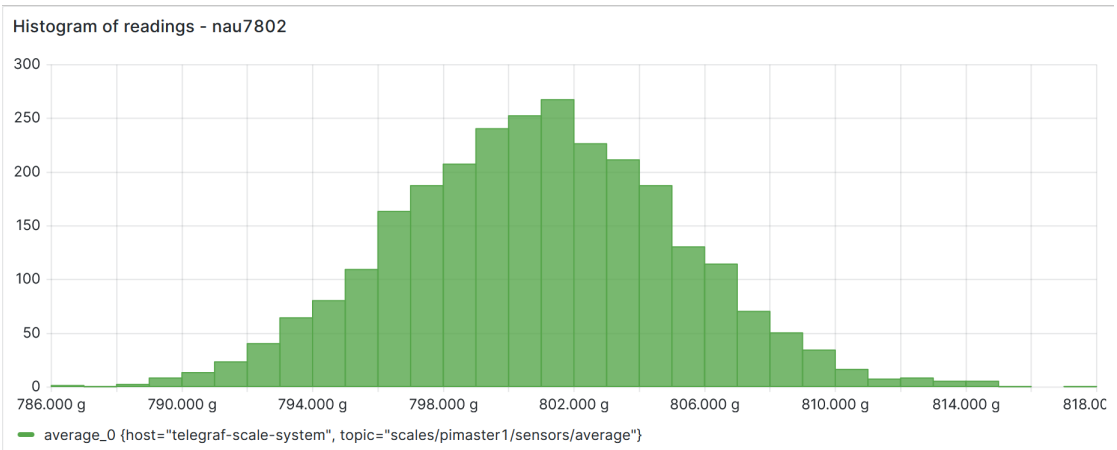
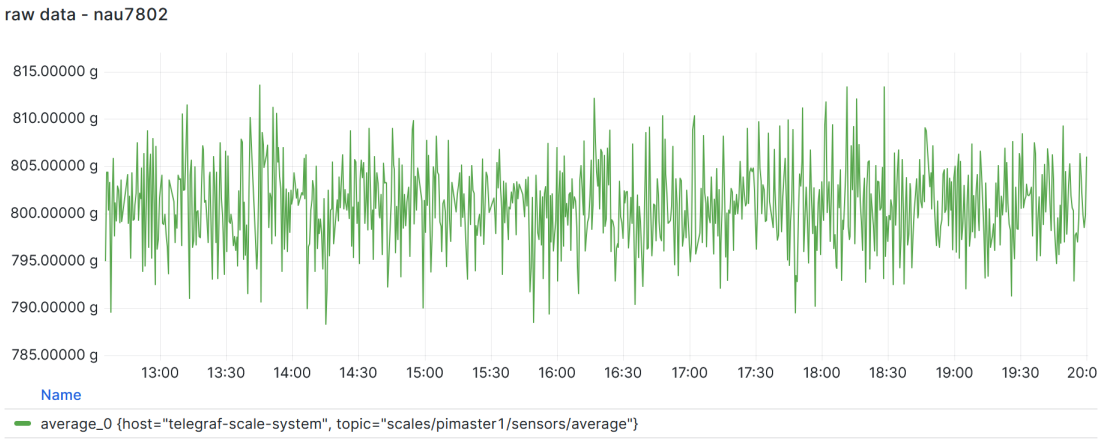


Figure 9.3: Histogram of readings using Slave_nau7802 front ends

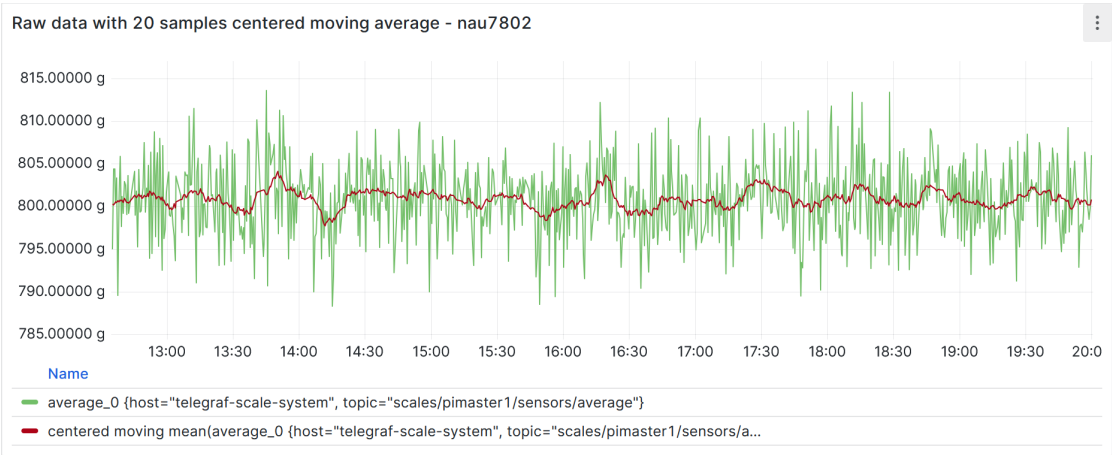
A focus on a shorter section of the measurement is provided in the time series of Figure 9.4a. From the image, it is clear that the standard deviation is just 4.32689 g, but outliers of more than ±10g are relatively frequent.

For the final application, there is no need to obtain data at a frequency as high as a measurement every 30 seconds, as the scale is currently configured. It is thus possible to apply a 20-sample (10 minutes) moving average in post-processing. The results are shown in Figure 9.4b with the red line over the raw data in green.

The averaged data maintains a fairly similar mean of 800.86188 g to the raw data, but the readings are much cleaner with a standard deviation reduced to 933.51257 mg over the total 2750 data points.



(a) Raw data



(b) Raw data with 20 samples moving average

Figure 9.4: Comparison of raw Slave_nau7802 data and the same data processed with a moving average filter.

Afterward, the second analog front-end, Slave_zssc3224, was tested.

Figure 9.5 shows the histogram of the 2750 measurements captured. Comparing Figure 9.3 and Figure 9.5, it is immediately clear that Slave_zssc3224 guarantees better performance than Slave_nau7802.

This second front-end produces a raw mean of 800.25573 g with a standard deviation of just 620.22182 mg. Similarly to the first case, the skewness of -0.02065 is low, thus the histogram is fairly symmetric, indicating the random nature of the error.

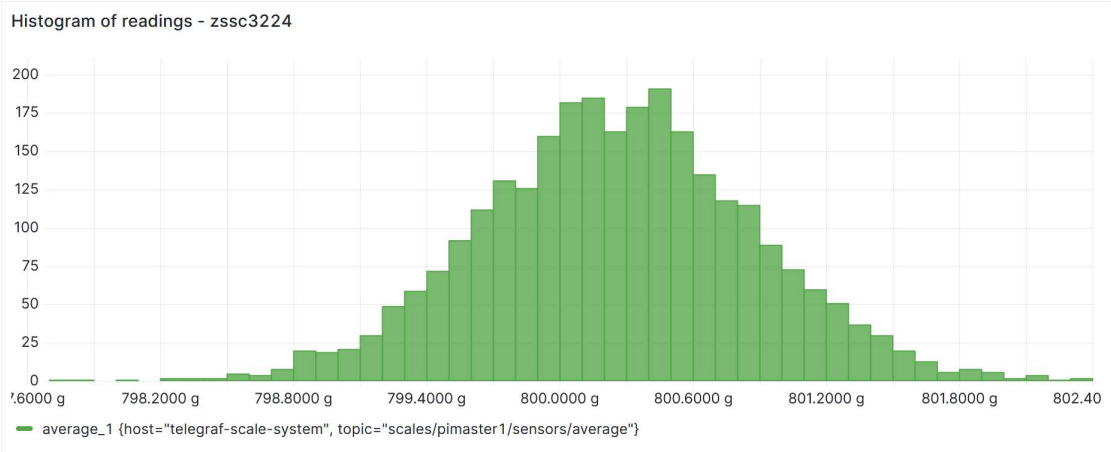


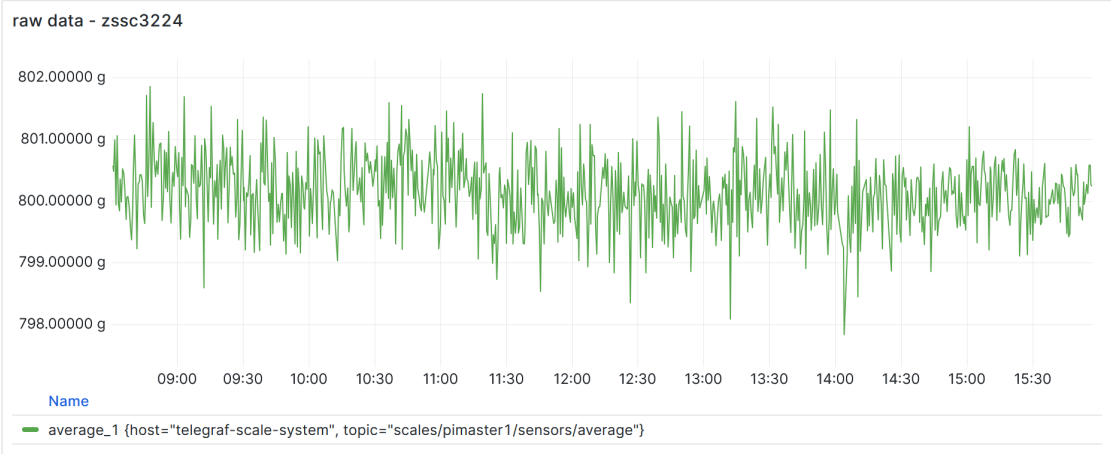
Figure 9.5: Histogram of readings using Slave_zssc3224 front ends

The standard deviation is almost one order of magnitude better in the second front-end compared to the first one.

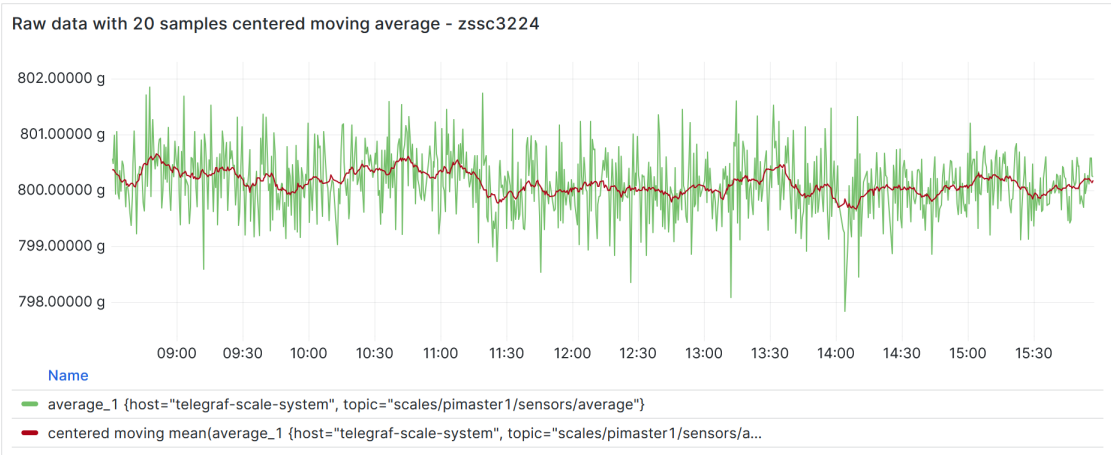
Figure 9.6a is a time series of the raw data acquired using the Slave_zssc3224 front-end. A 20-sample moving average was again applied to the data in post-processing.

Once post-processed, the results are shown with the red line over the green raw data in Figure 9.6a. The average of the processed data over the total 2750 samples is 800.25573 g with a standard deviation of 277.58580 mg

The results obtained in the performance validation phase are already adequate for both front-ends with the 50 kg load cell. Once the 10 kg load cells are available, with their greater output for an equivalent load, it is expected that a fivefold improvement of the results just described will be achieved.



(a) Raw data



(b) Raw data with 20 samples moving average

Figure 9.6: Comparison of raw Slave_zssc3224 data and the same data processed with a moving average filter.

Chapter 10

Final considerations

10.1 Economic considerations

Building a custom solution gives the flexibility to design a solution that fits the exact needs of the application, an advantage that a fully commercial solution simply can not achieve.

At the same time, when searching for a solution to a problem, costs are a pivoting point in the decision between a commercial, or embracing the development of a custom solution.

This section is devoted to the analysis of two commercial alternatives with similar specifications and similar performances to the solution developed in this thesis. The first solution embraces an approach typically more used in an industrial automation setting to create a partially custom solution, while the second approach is based on an entirely commercial system.

To obtain the results of chapter 9, rather than the 10 kg load cell that will be used in the final implementation, a 50 kg one was used. To keep the comparison fair, the alternative system analyzed in this section will have a similar capacity of around 50 kg.

The final objective of the two competitive systems is to send the data to the cloud infrastructure described in this thesis, disregarding in the calculations the costs of optional accessories such as irrigation control.

10.1.1 First competitor: industrial automation approach

This first alternative uses components and standards that are generally common in an industrial setting. This approach can be described as semi-custom since all the components are commercially sourced, finished elements that communicate using industry-standard protocols. These components can be used as building blocks for complex and modular systems.

The first element of the system is the load cell and scale platform. Given the excellent price-performance ratio, it was chosen to use the same as in the project of this thesis (LP7611 in 300mmx300mm size using L6N-C3-50kg load cell).

The analog signals are digitized using a dedicated module always built by Celmi, model "Data"[33].

Figure 10.1: Celmi Data load cell interface [33]

This integrated module connects to a single load cell, outputting the reading on standard digital interfaces, including RS-232 and RS-485. To digitize the four load cells required, it is necessary to use four independent modules.

Celmi Data is capable of digitizing a load cell with a linearity better than 0.01% of full scale, translating to theoretical performance slightly worse than the custom system using Slave_nau7802 analog front end.

Using Modbus RTU over the available RS-485 output port, all the modules can transmit the reading to Teltonika RUT906 IoT gateway. (Figure 10.2)

The Teltonika RUT906 can act as a Modbus-to-MQTT gateway, allowing data and commands to be sent and received from Modbus devices through the MQTT broker [34]. It is, thus, possible to connect the obtained system to the same cloud

Figure 10.2: Teltonika RUT906 IoT gateway [34]

infrastructure as the custom solution developed for this thesis. This solution has the advantage, over a custom one, of having the capability to transmit not just over WiFi and Ethernet but also over a cellular network.

This solution does not fully qualify with the requirement of using a Raspberry Pi SBC, but this might be irrelevant since the approach chosen is very flexible and allows for easy expansion with commercially available components. For example, relay boards featuring Modbus on RS-485 are commercially widely available.

The last element of the system is the power supply. RUT906 includes one, but the scale interface requires a 24-volt power supply. The absolute maximum power draw of each module is 3 W, adding to a total of 12 W for all the modules combined. Since all the components can be mounted to a DIN rail, a power supply that can be mounted to a DIN rail was selected. Mean Well HDR-15-24, 24 V 15 W power supply is an ideal candidate.

10.1.2 Second competitor: fully commercial solution

The second proposed alternative is a system that can be considered a full commercial solution.

Kern & Sohn is one of the leading companies in the production of high-quality laboratory, industrial, and medical scales. Their IoT-Line includes the scale model KGP 60K-3 10.3. This is a platform scale very similar to the one chosen for the project of this thesis. It features a maximum load of 60 kg with a rated linearity of reading of ± 0,008 kg [35]. The signals from the load cells are digitized by the included read box and transmitted over WiFi. There is an Ethernet version too that might cost a few tens of euros less, but actual pricing does not seem to be available at the moment of writing. To keep the comparison fair with the other solutions, only the WiFi was considered.

Figure 10.3: Kern & Sohn model KGP 60K-3 IoT scale [35]

Kern KGP 60K-3 communicates using proprietary standards and software. To interface it to the server using MQTT, it is necessary to have a dedicated computer that acts as a gateway. A Raspberry Pi could be a good fit. The Raspberry Pi could also be used to extend its functionalities, for example, to control irrigation, but this would not be appropriate for a proposal intended to be a ready-to-go commercial solution.

10.1.3 Comparison of the four solutions

Table 10.1 and Table 10.2 show a rough estimation of the costs of the two solutions proposed in this thesis, plus the two commercial alternatives in subsections 10.1.1 and 10.1.2. In estimating the price, it was considered just the weighing system. Any additional functionality and the costs related to wiring and mechanical mounting, such as electrical boxes and fasteners, and screws, were disregarded from calculations.

The commercial solutions cost, for a complete 4-scale system, respectively 1735,29 € and 2627,82 €, while the custom solutions cost 721,86 € or 744,94 €. The commercial solution costs between 2.5 and 3.5 times the custom one, providing similar performances, proving the validity from an economic point of view of the custom solution.

In this comparison, it was also not considered the added advantage of the modularity of the custom solution. The addition of the irrigation control functionality cost just 16,51 € + VAT, the commercial solutions can not compare, from a cost point of view, for a similar functionality improvement.

First competitor: industrial automation approach				Second competitor: fully commercial solution			
Description	**Unit price (€)**	**Quantity**	**Sub-total (€)**	**Description**	**Unit price (€)**	**Quantity**	**Sub-total (€)**
Celmi LP7611 scales	110,00	4	440,00	KERN KGP 60K-3 scale with WiFi	638,00	4	2552.00
Celmi DATA load cell digitizer	270,00	4	1080,00	Raspberry Pi 4 4Gb SBC	59,70	1	59,70
Teltonika RUT906 IoT gateway	196,09	1	196,09	SD card 32GB	9,73	1	9,73
Mean Well HDR-15-24 power supply	19,23	1	19,23	Raspberry Pi power supply	6,39	1	6,39
TOTAL (€):			1735,29	**TOTAL (€):**			2627,82

Table 10.1: Costs (excluding VAT) of the alternative solution using the industrial automation approach and the fully commercial solution

Custom solution using Slave_nau7802				Custom solution using Slave_zssc3224			
Description	**Unit price (€)**	**Quantity**	**Sub-total (€)**	**Description**	**Unit price (€)**	**Quantity**	**Sub-total (€)**
Celmi LP7611 scales	110,00	4	440,00	Celmi LP7611 scales	110,00	4	440,00
Raspberry Pi 4 4Gb SBC	59,70	1	59,70	Raspberry Pi 4 4Gb SBC	59,70	1	59,70
SD card 32GB	9,73	1	9,73	SD card 32GB	9,73	1	9,73
Raspberry Pi power supply	6,39	1	6,39	Raspberry Pi power supply	6,39	1	6,39
Pi_master assembled PCB	64,64	1	64,64	Pi_master assembled PCB	64,64	1	64,64
Slave_nau7802 assembled PCB	35,35	4	141,40	Slave_zssc3224 assembled PCB	41,12	4	164,48
TOTAL (€):			721,86	**TOTAL (€):**			744,94

Table 10.2: Costs (excluding VAT) of the custom solution using Slave_nau7802 and Slave_nau7802 front ends

10.2 Future improvements

The system proved to be generally behaving as expected and being functional, but a few improvements that a future revision might implement could further refine some aspects.

10.2.1 Scale system

The first improvement is purely software. The current implementation of the calibration algorithm is limited by the availability of just a single kit of calibration weights. Every channel has to be calibrated independently. It takes around 4 minutes for each channel. It is not a fully independent process because the operator has to add the calibration weights to obtain the second calibration point. Once one channel is calibrated, it is possible to continue with the following channel. This process is slow, around 15 minutes for a complete system, boring since the operator must maintain constant attention, and generally not very good, since the integration time is limited to 100 seconds for each weight point, leading to some calibrations being invalid.

An improved calibration algorithm would still be 15 minutes long, but calibrating all the scales in parallel using four calibration weights. The longer integration time improves the quality of the result while limiting the human intervention in the process, improving all aspects of the calibration process.

The second improvement is about the storage of the calibration coefficients. In the current implementation, these coefficients are stored in a CSV file on the Raspberry Pi computer managing each scale system. Each parameter refers to a specific analog front-end-scale combo, identified by the Pi_master channel it is connected to. The solution is adequate if there is no intention of swapping the position of the scale or of the sensor connected to each channel. If a small I2C memory IC is added to each analog front end interface, it could be used to store the calibration parameters independent of the position the specific scale is connected.

Other than this, the remaining improvements are minor things, such as substituting the red LED used with green ones that produce less photosynthetically active light.

10.2.2 Irrigation board

Currently, the irrigation is managed using the board explained in chapter 6. The proposed solution works very well, but it can be improved. First of all, it is necessary to modify the relay board, adapting it to the Adafruit breakout board to interface it to the Stemma QT I2C interface. The usage of a relay introduces additional mechanical components with limited reliability, and the wiring of the pumps is complex. The circuit to power the pumps is shown in Figure 10.5a. The relay only opens or closes a single contact. This means that all the negative terminals from all the pumps have to be tied together externally.

Adafruit, the same manufacturer of the I2C GPIO breakout board used, produces a second board advertised as *solenoid driver*. (Figure 10.4)

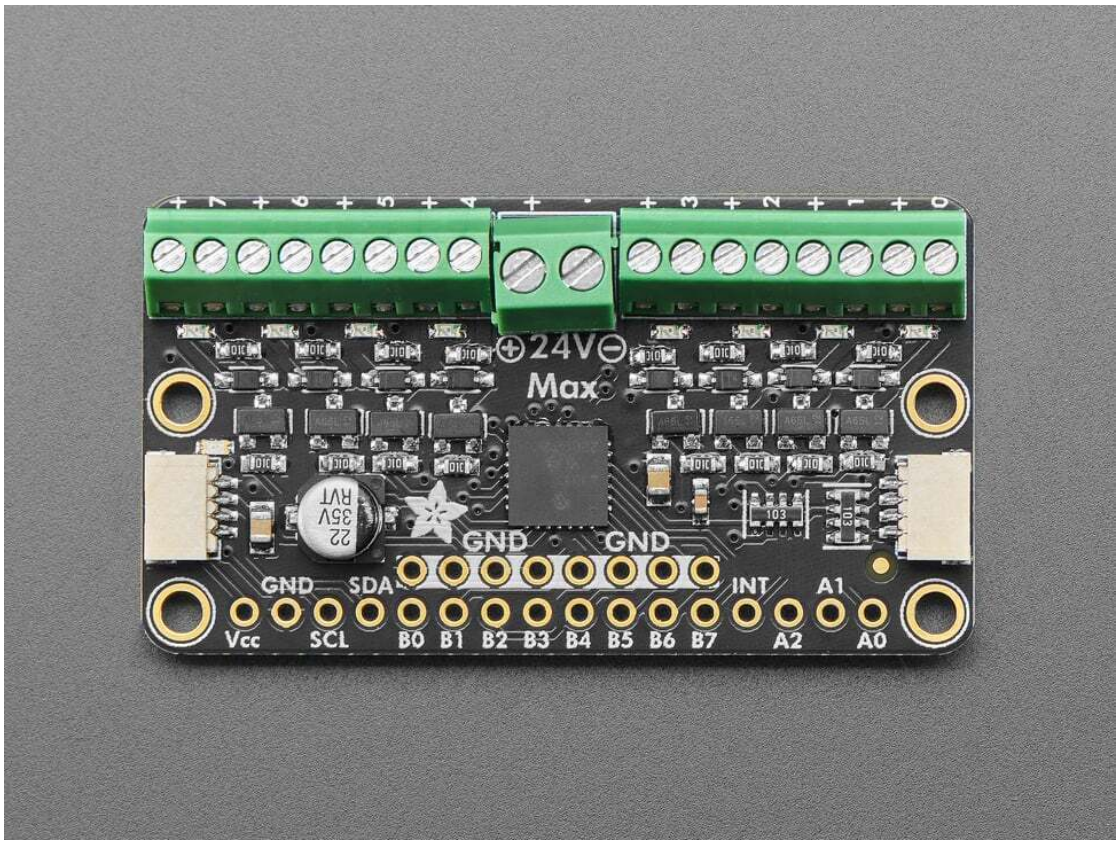


Figure 10.4: Adafruit I2C to 8 Channel Solenoid Driver [36]

The solenoid driver features eight N-Channel MOSFETS, one for each channel, used as load switches. These MOSFETS are directly connected to MCP23017 I2C GPIO, the same chip at the heart of the breakout board used in the relay board explained in chapter 6. Every MOSFET features diode protection and is capable

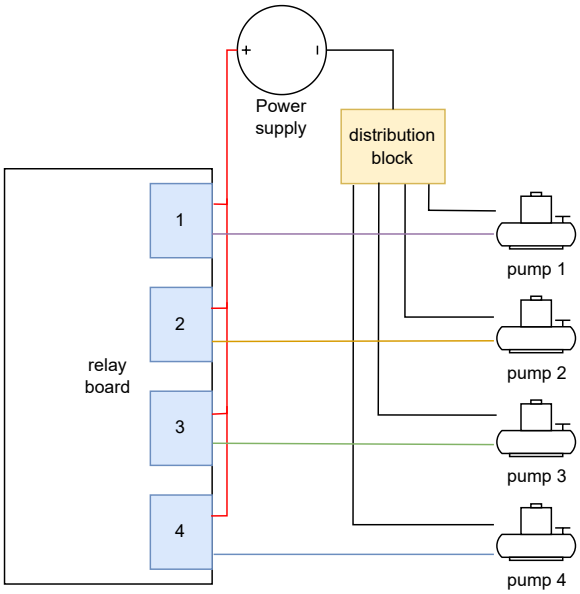
of directly controlling inductive loads.

The board also acts as a power distribution board. Every channel features two screw terminals, one for the positive and one for the negative. All the positive terminals are internally shorted on the PCB, while the negative terminals are switched by the MOSFET. The power supply for all the motors is connected to the central, bigger terminals. The features just listed greatly simplify the wiring for all the motors. An example of the simplified wiring is in Figure 10.5b.

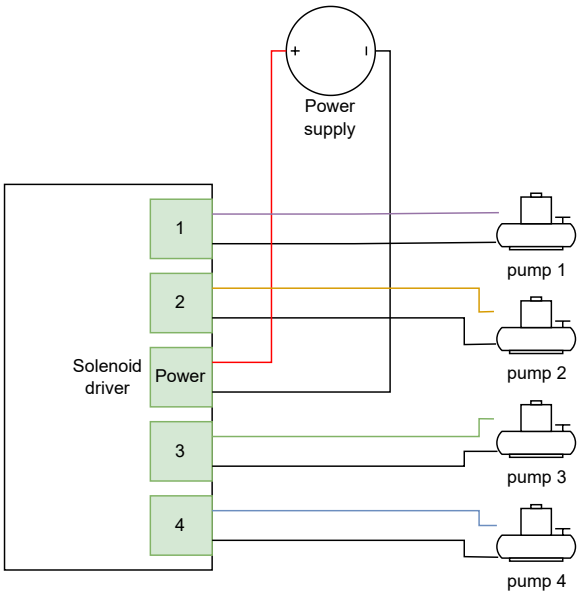
Advantages in the wiring are also on the digital communication interface because the board features STEMMA QT interface as standard.

This board is also smaller and cheaper, 12,74 € + VAT, compared to the 16,51 € + VAT of the current solution.

The only reason why this solenoid driver was not used for this thesis in favor of the relay board is that the solenoid driver is so new that it is not yet possible to find available stocks to purchase it.



(a) Wiring using relay board



(b) Wiring using solenoid driver

Figure 10.5: Comparison of pump wiring diagram using the two different boards

Chapter 11

Conclusion

In chapter 1, the requirements for a system based on a Raspberry Pi capable of monitoring the water usage and vegetative growth of plants in growth chambers are explained. From this conceptual foundation, in the following chapters, a full custom hardware and software system was created to meet the objectives. Throughout the entire design process, the utmost importance was given to system reliability and ease of deployment.

The modular approach chosen allows easy further expansion of the system with improved capabilities and measurements, transforming it into the possible backbone of a more complex monitoring and automation system for growth chambers.

Performance validation proves that the proposal can be a more capable and flexible solution compared to commercial alternatives while costing a fraction of the price.

To conclude, the results obtained confirm the validity of the system as a flexible, cost-effective, easily deployable, and easily maintained solution, ready to be integrated into the monitoring systems of growth chambers supporting the research of Politecnico di Torino.

Appendix A

ZSSC3224 adress definition code

The code in Listing A.1 is used to set the I2C address of ZSSC3224.

A full reboot of the sensor is necessary after an address change.

```python
import time
from smbus2 import i2c_msg, SMBus

import time

bus = SMBus(1)

try:

    bus.write_byte(0x00, 0x02)
    #bus.write_byte(0x32, 0x02)
    time.sleep(0.02)

    read = i2c_msg.read(0x00, 3)
    #read = i2c_msg.read(0x32, 3)
    bus.i2c_rdwr(read)

    bus.write_i2c_block_data(0x00, 0x42, [0x00, 0x32])
    time.sleep(0.02)

except KeyboardInterrupt:
    print("\n[SHUTDOWN]")
```

Listing A.1: Function to set the I2C address of ZSSC3224

Appendix B

MQTT configuration messages examples

```
{
  "action": "calibrate",
  "sensor_id": 2,
  "calibration_weight": 2.0
}
```

Listing B.1: Calibration command example

Listing B.1 is an example of a calibration request payload to calibrate channel 2 with a calibration weight of 2.0 kg.

This message has to be sent to *scales/{HOSTNAME}/device/commands*

```
[
  {"time": "10:18", "sensor": 1, "duration": 50},
  {"time": "08:05", "sensor": 2, "duration": 40},
  {"time": "18:00", "sensor": 3, "duration": 30},
  {"time": "18:10", "sensor": 4, "duration": 60}
]
```

Listing B.2: Irrigation schedule change command example

This second message in Listing B.2 is an example of an update to the irrigation schedule. All the schedules have to be updated at the same time. The duration is expressed in seconds.

This message has to be sent to *scales/{HOSTNAME}/device/irrigation*

Appendix C

Usage of python-scales-installer.sh

1. Prepare a USB stick. In the USB stick, you must place the *python-scales-installer.sh* file and create a folder called *python-scales-project* containing all the files of the Python project.
2. Insert the USB stick in the Raspberry Pi.
3. Connect in SSH to the Raspberry Pi.
4. Navigate to */media*.
5. Navigate to the folder named after the user. For example, if the user is *admin*, the folder is *./admin*.
6. Navigate to the folder corresponding to the connected USB stick.
7. Navigate to the folder where the installation script and Python project are saved.
8. Execute the installer script with: *sudo bash ./python-scales-installer.sh*
9. Check if everything is working with: *sudo journalctl -u python-scales.service -f*

Bibliography

[1] *Plantzcare website*. Available here. Plantzcare (cit. on p. 1).

[2] World Economic Forum. *Top 10 Emerging Technologies of 2023*. Technical Report. World Economic Forum, June 2023 (cit. on p. 2).

[3] Stefano Calvo, Mattia Barezzi, Danilo Demarchi, and Umberto Garlando. «In-vivo proximal monitoring system for plant water stress and biological activity based on stem electrical impedance». In: *2023 9TH INTERNATIONAL WORKSHOP ON ADVANCES IN SENSORS AND INTERFACES, IWASI*. International Workshop on Advances in Sensors and Interface. 9th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI), Monopoli, ITALY, JUN 08-09, 2023. IEEE; IEEE Elect Packaging Soc; Politecnico Bari; Ist Nazl Fis Nucl; Dipartimento Ingn Elettrica Informazione; Politecnico Bari, Dipartimento Interateneo Fis; Univ Bari, Dipartimento Interateneo Fis; Puglia Promozione; G Tec Med Engn GmbH. 2023, pp. 80–85. ISBN: 979-8-3503-3694-8. DOI: 10.1109/IWASI58316.2023.10164553 (cit. on p. 2).

[4] Megh Raj Goyal and Eric W. Harmsen. *Evapotranspiration: Principles and Applications for Water Management*. English. First edition. Toronto: Apple Academic Press, 2014. ISBN: 978-0-429-16135-5 (cit. on p. 2).

[5] Dalyn McCauley, Alexander Levin, and Lloyd Nackley. «Reviewing Minilysimeter Controlled Irrigation in Container Crop Systems». In: *HORTTECHNOLOGY* 31.6 (Dec. 2021), pp. 634–641. ISSN: 1063-0198. DOI: 10.21273/HORTTECH04826-21 (cit. on p. 2).

[6] *What You Need To Know About Strain Measurement Errors*. Available here. Willrich Precision Instrument (cit. on p. 6).

[7] *Measuring Strain with Strain Gages*. Available here. NATIONAL INSTRUMENTS CORP (cit. on pp. 7, 8).

[8] *Quarter Bridge, Half Bridge and Full Wheatstone Bridge Strain Gauge Load Cell configurations*. Available here. Transducer Techniques (cit. on pp. 7, 8).

[9] *NAU7802 data sheet*. https://www.nuvoton.com/export/resource-files/en-us--DS_NAU7802_DataSheet_EN_Rev2.6.pdf. Hsinchu Science Park, Taiwan: Nuvoton (cit. on pp. 9, 11, 20, 28, 32, 34, 82).

[10] *L6N data sheet*. https://www.zemiceurope.com/media/Documentation/L6N_Datasheet.pdf. Chang'an District, Xi'an Shaanxi China: Zemic (cit. on pp. 9, 31, 49).

[11] *ADC Architectures III: Sigma-Delta ADC Basics (MT-022)*. Available here. One Analog Way Wilmington, MA 01887: Analog Devices (cit. on pp. 10, 11).

[12] *delta-sigma-adcs*. Available here. Monolithic Power Systems (cit. on p. 12).

[13] *ZSSC3224 data sheet*. https://www.renesas.com/en/document/dst/zssc3224-datasheet?srsltid=AfmBOoorbhn515MTcTGRe7j-u6K0-RZomB-F_c145RGAexCmlgbX1pbB. TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan: Renesas Electronics (cit. on pp. 13, 14, 20, 47, 48, 83, 84).

[14] *I2C-bus specification and user manuals (UM10204)*. Available here. High Tech Campus 60, 5656 AG Eindhoven, Netherlands: NXP Semiconductors (cit. on p. 15).

[15] *PCA9615 data sheet*. https://www.nxp.com/docs/en/data-sheet/PCA9615.pdf. High Tech Campus 60, 5656 AG Eindhoven, Netherlands: NXP Semiconductors (cit. on pp. 16, 34, 50, 56, 60).

[16] *Raspberry Pi 4 model B official webpage*. Available here. Raspberry Pi (cit. on p. 18).

[17] *Adafruit STEMMA and STEMMA QT, what is STEMMA QT*. Available here. Adafruit (cit. on p. 18).

[18] *MAX7320 data sheet*. https://www.analog.com/media/en/technical-documentation/data-sheets/max7320.pdf. One Analog Way Wilmington, MA 01887, USA: Analog Devices (cit. on pp. 19, 20).

[19] *MCP23017 data sheet*. https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP23017-Data-Sheet-DS20001952.pdf. Chandler, Arizona, United States: Microchip (cit. on pp. 22, 65).

[20] *MQTT Version 3.1.1*. Available here. OASIS, Dec. 2015 (cit. on pp. 22, 23).

[21] *Eclipse mosquitto logo*. Available here. Eclipse (cit. on p. 24).

[22] Texas Instruments. *Analog Engineer's Pocket Reference Guide*. Texas Instruments, 2019 (cit. on p. 28).

[23] *SimSurfing*. Available here. Murata (cit. on pp. 28, 29).

[24] Rajan Arora. *I2C Bus Pullup Resistor Calculation*. Application Report SLVA689. Texas Instruments, Feb. 2015 (cit. on p. 34).

[25] European Parliament and Council of the European Union. *Directive 2014/30/EU of the European Parliament and of the Council of 26 February 2014 on the harmonisation of the laws of the Member States relating to electromagnetic compatibility (recast)*. https://eur-lex.europa.eu/eli/dir/2014/30/oj. Feb. 2014. URL: https://eur-lex.europa.eu/eli/dir/2014/30/oj (cit. on p. 38).

[26] European Commission. *Commission Notice — The 'Blue Guide' on the Implementation of EU Product Rules 2022*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=oj:JOC_2022_247_R_0001. July 2022. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=oj:JOC_2022_247_R_0001 (cit. on p. 39).

[27] *MIC5528 data sheet*. https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/MIC5528-High-Performance-500mA-LDO-in-Thin-and-Extra-Thin-DFN-Packages-DS20005982B.pdf. Chandler, Arizona, United States: Microchip (cit. on pp. 48, 49).

[28] *TCA9517 data sheet*. https://www.ti.com/lit/ds/symlink/tca9517.pdf. Dallas, Texas, United States: Texas Instruments (cit. on p. 60).

[29] *Adafruit pump characteristics*. Available here. Adafruit (cit. on pp. 64, 65).

[30] *Adafruit i/o expander*. Available here. Adafruit (cit. on p. 67).

[31] *Adafruit i/o expander circuit diagram*. Available here. Adafruit (cit. on p. 67).

[32] *Grafana visualizations documentation*. Available here. Grafana (cit. on p. 131).

[33] *DATA data sheet*. https://www.celmi.com/wp-content/uploads/2022/08/data_ita_21779.pdf. L.go Brugnatelli 13/16 – 20090 Buccinasco (MI) Italy: Celmi (cit. on p. 141).

[34] *RUT906 data sheet*. https://www.teltonika-networks.com/_files/ugd/814ec4_98637ca9430e42d993377be94327639f.pdf. K. Baršausko st. 66, LT-51436, Kaunas, Lithuania: Teltonika (cit. on pp. 141, 142).

[35] *KGP 60K-3 product page*. Available here. Kern Sohn (cit. on p. 143).

[36] *Adafruit I2C to 8 Channel Solenoid Driver*. Available here. Adafruit (cit. on p. 148).