Autonomous Plant Monitoring Station IoT Project Report

Giulia Titton, *240311*

***Abstract*—Microbial Fuel Cells (MFCs) are bio-electrochemical devices that generate electricity thanks to the metabolic activity of bacteria. Leveraging this capability, we have developed an autonomous and standalone monitoring system for a plant station. The monitoring system is critical for ensuring the health and efficiency of bacteria which directly impacts energy production. Our system includes an analog-to-digital converter installed near the plant station (AD7606), which interfaces with a Nucleo-STM32F401RE board via SPI. This board periodically transmits data through UART to a Raspberry Pi 4, which hosts a Node-RED server for data processing, InfluxDB for data storage, and Grafana for data visualization. This comprehensive setup enables real-time monitoring and analysis, ensuring optimal performance of the MFCs and the plant station. The implementation of this monitoring system not only enhances the efficiency of MFCs but also provides valuable insights into their operational dynamics, potentially guiding future advancements in scalable and sustainable self-powered monitoring solutions in various applications.**

I. Introduction

The Internet of Things (IoT) is a network of interconnected devices that exchange data with each other and the cloud. These devices, which can range from household items to industrial tools, are equipped with sensors and software. They enable organizations across various industries to operate more efficiently, enhance customer service, improve decision-making, and increase business value. IoT facilitates data transfer over networks without requiring direct human interaction, connecting devices such as heart monitors, agricultural biochips, and smart vehicles to transmit data seamlessly.
The Internet of Things (IoT) operates through interconnected smart devices equipped with sensors and embedded systems. These devices gather data from their environments, which is then transmitted to IoT gateways for analysis by applications or back-end systems. Key components of an IoT ecosystem include web-enabled devices that use processors, sensors, and communication hardware to collect and transmit data. Connectivity enables these devices to communicate over internet networks, utilizing IoT gateways as central hubs for data aggregation. Data analysis focuses on extracting meaningful insights and addressing potential issues locally to optimize bandwidth consumption. Human interaction with IoT devices involves setup, instructions, and data access via graphical user interfaces such as websites or mobile apps. IoT systems also leverage connectivity protocols and may incorporate artificial intelligence and machine learning to enhance data processing capabilities.
Moreover, IoT revolutionizes both consumer lifestyles and business operations by integrating smart devices. Consumers automate home settings with devices like smartwatches and thermostats, while businesses gain real-time insights into operations, optimizing efficiency and reducing costs. IoT enables automation, cuts down on waste, and improves service delivery, making manufacturing and logistics more transparent and cost-effective. Various industries, spanning from manufacturing to agriculture, experience benefits that drive digital transformation and enhance competitiveness. For example, in agriculture, it simplifies farming through data collection on weather conditions and soil quality, automating processes. In construction, IoT monitors infrastructure integrity, detecting potential safety issues and improving incident response while reducing operational costs. For home automation, IoT enables remote control and automation of mechanical and electrical systems, enhancing comfort, energy efficiency, and security through devices like smart thermostats, lighting systems, and voice assistants like Alexa and Siri. In transportation, IoT devices enhance vehicle performance monitoring, optimize routes, and track shipments efficiently, reducing fuel costs and ensuring cargo arrives in optimal condition. Wearable devices with sensors simplify daily tasks by analyzing user data and enhancing public safety, providing optimized routes for emergency responders and monitoring vital signs in hazardous environments. Lastly, in energy management, IoT-enabled systems like smart grids and meters optimize energy usage, implement demand-response programs, and integrate renewable energy efficiently by analyzing data to identify usage patterns and peak demand times.
Every coin has two sides, and the Internet of Things (IoT) is no exception. While IoT devices offer benefits such as easy access to real-time information, enhanced communication efficiency, and cost savings through automation, they also present challenges. With more devices connected, security risks increase. Managing devices becomes more complex, and there's a risk of device malfunctions. Compatibility issues and different platforms can make it harder to integrate systems. Automating tasks might replace jobs, and following varied regulations can be challenging for businesses using IoT [1].
In the world of IoT security, there are significant challenges. These include issues with encryption, lack of thorough testing, and vulnerabilities such as default passwords and IoT-specific malware. These problems lead to risks like network attacks, insecure data transmission, and concerns about privacy. To tackle these challenges, strong security measures such as encryption, secure authentication, and regular updates are crucial to protect IoT devices and networks.
In IoT design, ensuring different devices work together smoothly (interoperability) is a major challenge. Security remains a top concern, covering device, network, and data protection, as well as privacy safeguards. Scalability is also

critical, requiring effective management of data and network capacity to handle the growing number of connected devices. To overcome these challenges, scalable technologies and smart data management strategies are essential to maintain performance and efficiency as IoT systems expand.

Reliability is also crucial to ensure devices perform consistently without failure, addressing challenges like device malfunctions and network stability. Power consumption is critical due to limited battery life, requiring energy-efficient designs and effective management. Privacy concerns arise from data collection and transmission, demanding secure practices and technologies like encryption. Designers also navigate challenges such as device size, cost constraints, and robust security measures throughout development and deployment to create effective IoT systems.

Lastly, deploying Internet of Things systems comes with challenges like connectivity issues, cross-platform compatibility, data collection and processing complexities, and the need for skilled resources. Integration with existing technology, building robust network infrastructure, efficient device and data management, and ensuring security are also critical. Cost considerations must balance deployment expenses with long-term benefits. To overcome these challenges, organizations should adopt a structured approach, selecting compatible hardware and software, planning network infrastructure carefully, implementing strong security measures, and optimizing resource allocation for maximum efficiency and return on investment [2].

This paper focuses on IoT infrastructure for monitoring plant health using distributed sensors such as humidity and temperature sensors. The foundation of this approach lies in the Microbial Fuel Cell, described in section III-A, which provides the essential data for our analysis. We collect, process and store this data in the IoT system in order to provide real-time insights and actionable information for optimizing plant health management. In fact, we aim to monitor environmental temperature, soil humidity, and the voltage provided by the MFC. These factors collectively contribute to the comprehensive assessment of plant health.

To clearly outline the contributions of this project, we have identified several key areas of impact:

- IoT Integration: Implementation of a robust IoT infrastructure for real-time monitoring of plant health parameters.
- Microbial Fuel Cell: Leveraging data from Microbial Fuel Cells to gain insights into soil conditions and plant health.
- Comprehensive Environmental Monitoring: Continuous monitoring of environmental temperature and soil humidity to assess and predict plant health.
- Data Collection and Analysis: Efficient data collection, processing, and storage to facilitate informed decision-making.
- Enhanced Plant Health Management: Providing actionable data to improve plant health management practices.

The rest of the report is organized as follows. Related Works is reviewed in Section II. Section III presents the technology used; we describe the hardware and software developed in the project with a special focus on each component of the IoT infrastructure. Section IV shows some of the issues encountered during the development process of the project and their solution. Section V presents some real data acquired while testing the system. Lastly, Section VI concludes the report and offers ideas about future works.

II. Related Work

In this section, we review existing research and projects related to our work. By examining these studies, we can highlight the gaps in current knowledge and demonstrate how our project builds upon and contributes to the existing body of work.

One notable study in the realm of IoT-based plant monitoring is by Puja A. Chavan et al. (2022) [3], which presents the design and implementation of a Smart Plant Monitoring System.In their paper, the authors tackle the issue of neglecting urban household plants due to busy schedules. They employ IoT technology to address this challenge, integrating a soil moisture sensor, temperature and humidity sensor, relay module, Wi-Fi module in NodeMCU (ESP8266), and the Blynk app. This system monitors and manages plant care by transmitting data on soil moisture, temperature, and humidity to the Blynk app. The automated watering solution aims to optimize plant growth while conserving water, enabling users to receive remote notifications about their plant's watering needs. The paper also presents a comprehensive Blynk app interface, flowcharts demonstrating the operation of the DC motor pump based on soil moisture levels, and an overview of the Smart Plant Managing System using IoT. Figure **??** illustrates the complete hardware system developed in their study. This study forms a crucial foundation for our work, particularly in its use of distributed sensors for environmental monitoring and its focus on automated plant care solutions. We aim to further enhance plant health monitoring through the integration of microbial fuel cells (MFC) for additional data insights.

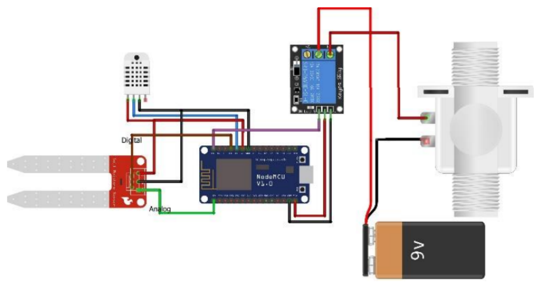


Figure 1: System proposed by [3].

In addition to the approach presented in the first paper, the second paper expands on the application of IoT in smart plant monitoring systems, exploring the usage of Raspberry Pi. Even if it is not recent (2018), the technology used remain current and relevant [4]. The proposed work aims to develop an Embedded System for plant monitoring and watering using IoT, with Raspberry Pi Model B 3.1 as the processor and various sensors for environmental sensing. The hardware includes a

Temperature sensor (LM35), Humidity sensor, Moisture sensor, Light sensor, and IR sensor. Additionally, it incorporates a Relay and motor for automated watering based on soil moisture levels. The system is supported by a server with a database and an IoT-enabled web-based application, which displays real-time sensor data, facilitating remote monitoring via smartphones or direct connections to the system. Figure 2 presents the complete IoT infrastructure.

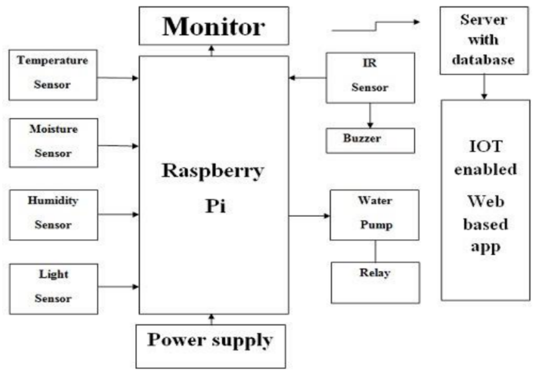


Figure 2: System proposed by [4].

A third paper [5] introduces an IoT-based Smart Plant Monitoring System designed to oversee plant growth and health. This system leverages various sensors to gather environmental data crucial for plant maintenance. It begins by collecting data from a DHT-11 sensor, which measures temperature and humidity levels in the surroundings. This data is processed by an Arduino microcontroller, which then transmits it to the Blynk application via an ESP8266 Wi-Fi module. In real-time, the Blynk app displays the temperature and humidity data, allowing users to receive notifications if these parameters exceed predefined thresholds. Moreover, the system features a solenoid valve controlled by the Arduino to automate watering based on sensor readings. A relay manages the valve's operation, further enhancing the system's automation capabilities. Overall, this IoT-based solution empowers users with remote monitoring and control, optimizing plant environments while reducing manual intervention.
As last paper we analyse [6]. This paper addresses challenges faced by farmers such as unpredictable weather and plant diseases through IoT and machine learning in agriculture. It employs sensors like soil moisture, humidity-temperature, and a camera module for disease detection in tomato plants. Utilizing Azure Custom Vision Model enhances disease detection accuracy, while the ThingSpeak platform displays real-time environmental data. The architecture includes two modules: plant health monitoring and disease detection. The former monitors conditions like air temperature, humidity, soil temperature, and moisture, adjusting watering via a relay based on sensor readings. This setup ensures optimal plant care without over- or under-watering, supported by a user-friendly GUI dashboard. The components used in this system include DHT11 (Air Temperature and Humidity Sensor), DS18B20 (Soil Temperature Sensor), Capacitive Soil Moisture Sensor, SI114X (Sunlight Sensor), and a Relay (Actuator). Figure 3 shows the complete architecture.

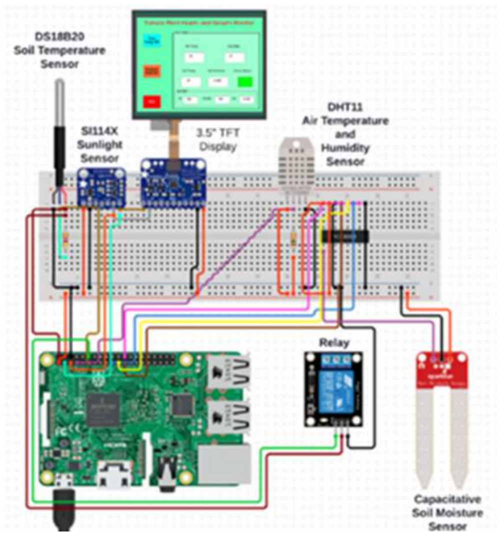

DS18B20 Soil Temperature Sensor
SI114X Sunlight Sensor
3.5" TFT Display
DHT11 Air Temperature and Humidity Sensor
Relay
Capacitative Soil Moisture Sensor

Figure 3: System proposed by [6].

To conclude this section, our approach offers several advantages over existing systems. Firstly, our use of the Raspberry Pi as a gateway provides robust processing power, enhancing system efficiency and reliability. Additionally, the AD7606 analog-to-digital converter delivers immediate and highly accurate data with its 16-bit resolution, ensuring precise monitoring of environmental parameters. Moreover, leveraging Node-RED as a powerful and user-friendly system facilitates seamless programming and offers an intuitive user interface for enhanced usability. Lastly, the local database data collection capability ensures optimal data management and accessibility. These features collectively contribute to the effectiveness and superiority of our proposed solution in plant monitoring and management.

III. Hardware setup

The general flow and idea of the project is described in Figure 4. Firstly, data is acquired from the MFC and converted from an analog value to a digital value thanks to the AD7606 module. The converted data is sent through SPI to the STM32 microcontroller on the Nucleo board which will elaborate and transfer the data to RaspberryPi employed as server. RaspberryPi will manage the internet connectivity to the overall IoT infrastructure, based on a Node-Red server, an InfluxDB database and Grafana. Moreover it will manage the actions to be taken according to the messages coming from Node-RED.
In the next subsections are described all the components of the system.

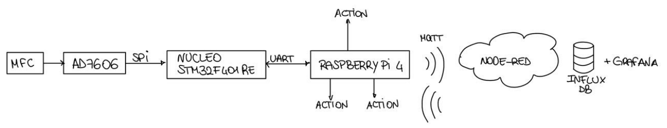


Figure 4: General working flow

A. Microbial fuel cells (MFCs)

Microbial fuel cells are systems that can generate electricity by harnessing microorganisms' metabolic activity [7]. Their construction is ambitious: they are composed by an anodic chamber, a cathodic chamber, and proton exchange membrane (PEM). The anodic chamber is maintained under anaerobic conditions, allowing microorganisms to generate electrons and protons, with carbon dioxide as a byproduct of oxidation. The anode absorbs the electrons, which are then transferred to the cathode through an external circuit. Protons move to the cathodic chamber via a proton exchange membrane (PEM). Continuous electricity generation is achieved by keeping microorganisms separate from oxygen in the anodic chamber. Various strategies, such as altering microorganisms, membranes, anodic surfaces, and bacterial genes, have been explored to enhance electricity generation [8]. Figure 5 shows the structure of the MFCs.

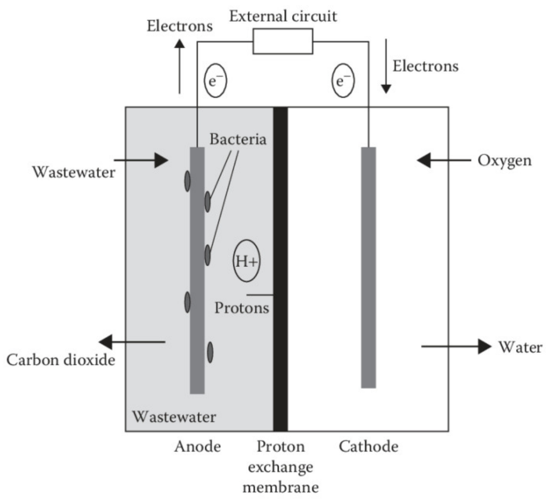


Figure 5: MFC structure

Fuel cells are exploited in various scenarios in real life. For example [8] employ MFCs for Sustainable Wastewater Treatment.

articoli per spiegare cosa solo le fuel cell: https://www.sciencedirect.com/topics/biochemistry-genetics-and-molecular-biology/microbial-fuel-cell

costruzione celle: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10223362/

B. AD7606

The AD7606 is an eight-channel, simultaneously sampled, 16-bit analog-to-digital converter (ADC) capable of sampling rates up to 200 kSPS per channel. It operates from a 5V supply and supports ±10V and ±5V bipolar input signals. Key features include on-chip low dropout regulators (LDOs), a reference and reference buffer, track and hold circuitry, supply conditioning circuitry, an on-chip conversion clock, oversampling capability, and both high-speed parallel and serial interfaces. The AD7606 has low noise, high input impedance amplifiers suitable for input frequencies of 5-10 kHz and incorporates a front-end anti-alias filter with 40 dB attenuation at 200 kSPS. Data acquisition and conversion are managed by CONVST signals and an internal oscillator, with the option for oversampling to enhance noise performance and reduce output code spread at lower throughput rates.

The communication with the STM32 is based on the Serial Peripheral Interface (SPI). It is a synchronous serial communication protocol primarily used for short-distance communication in embedded systems. Developed by Motorola, SPI is designed for high-speed data transfer between a master device and one or more slave devices. It operates in half or full duplex mode, allowing both single and simultaneous data transmission and reception. Four main signals are employed in this kind of communication:

- MOSI (Master Out Slave In): Carries data from the master to the slave;
- MISO (Master In Slave Out): Carries data from the slave to the master;
- SCLK (Serial Clock): Generated by the master to synchronize data transmission;
- SS/CS (Slave Select/Chip Select): Activated by the master to enable communication with a specific slave.

The main advantages of SPI are its simplicity, its high data transfer rates, and its ability to connect multiple slave devices using individual SS lines, features that allow to employ SPI in a wide range of applications, such as sensor interfacing (like in this case), SD card communication and display driving. On the counterpart, it lacks built-in error checking and does not support multi-master configurations. Figure 6 shows briefly how the simplest communication between master and slave works.

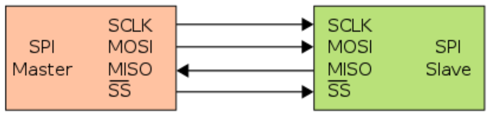


Figure 6: SPI protocol

In this application, the SPI protocol works as in Figure 7, according to the datasheet of AD7606. The AD7606 allows the simultaneous sampling of eight analog input channels. When the CONVST pins (CONVST A and CONVST B) are tied together, a single CONVST signal controls both inputs, and the rising edge of this signal initiates simultaneous sampling on all channels (V1 to V8). An on-chip oscillator performs the conversions with a conversion time (t_{CONV}) of 4 μs for all eight channels.

The BUSY signal indicates when conversions are in progress. When the rising edge of CONVST is applied, BUSY goes high and transitions low at the end of the conversion process. The falling edge of the BUSY signal switches all track-and-hold amplifiers back to track mode and indicates that new data can be read via the serial interface through the MISO pin.

Timing diagram and sequence of signals must be ensured for a correct conversion of data.

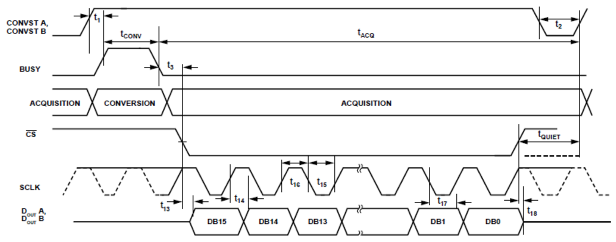


Figure 7: Serial Read Operation

Output data are stored and converted in 2's complement to be then converted again to get the final value, following the diagram in Figure 8 and using the Equation 1. The result of conversion is the resulting level of tension.

$$\text{ConvertedData} = \frac{\text{ReceivedData} \times (5 \times 2.5)}{32768 \times 2} \tag{1}$$

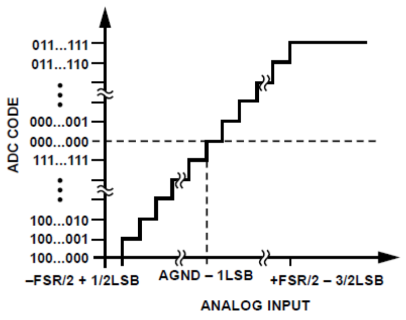
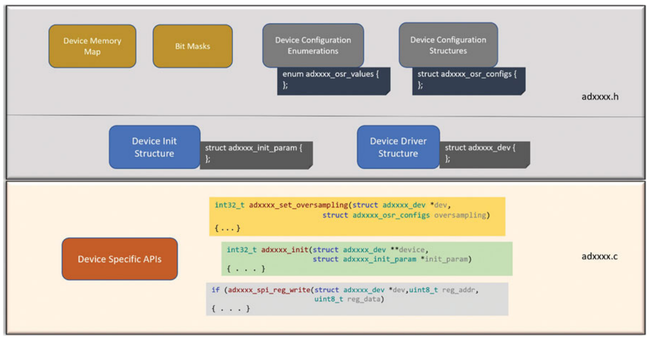


Figure 8: AD7606 Transfer Characteristic

C. Nucleo-F401RE and no-Os drivers

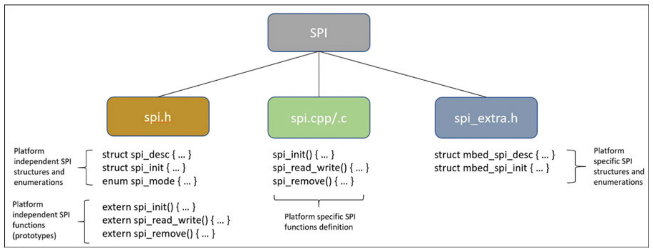
Nucleo-F401RE has been exploited to manage the data collection from AD7606. Analog Devices offers no-OS (no operating system) drivers and platform drivers, which are useful for building an application firmware with Analog Devices precision analog-to-digital converters and digital-to-analog converters, with a high level of performance in terms of speed, power, size, and resolution [9]. No-OS drivers are responsible for device configuration, data capture from the converter, performing the calibration, etc. No-OS drivers make use of the platform driver layer to allow the reuse of the same no-OS drivers across multiple hardware/software platforms, making the firmware highly portable. The use of a platform driver layer insulates the no-OS drivers from knowing about the low-level details of platform specific interfaces such as SPI, I2C, GPIO, etc., which makes the no-OS drivers reusable across multiple platforms without changing them. The structure of the no-Os driver is shown in Figure 9.



```
enum adxxxx_osr_values {
};
struct adxxxx_osr_configs {
};
struct adxxxx_init_param {
};
struct adxxxx_dev {
};
int32_t adxxxx_set_oversampling(struct adxxxx_dev *dev,
                                struct adxxxx_osr_configs oversampling)
{...}
int32_t adxxxx_init(struct adxxxx_dev **device,
                    struct adxxxx_init_param *init_param)
{ . . . }
if (adxxxx_spi_reg_write(struct adxxxx_dev *dev,uint8_t reg_addr,
                         uint8_t reg_data)
{ . . . }
```

Figure 9: Structure of no-Os driver

In the no-Os library, the files "ad7606.h" and "ad7606.c" are used to capture data from the AD7606 analog-to-digital converter. The header file (ad7606.h) includes the public programming interface with device-specific structures, enumerations, register addresses, and bit masks. The source file (ad7606.c) implements the interface for device initialization and removal, register read/write operations, data reading, and setting/getting device-specific parameters.
Talking about platform drivers, they are a type of hardware abstraction layer (HAL) that encapsulate platform-specific APIs. They are used by no-OS device drivers or user applications to ensure independence from the underlying hardware and software platforms. These drivers provide a standardized interface for low-level hardware functions, including SPI, I2C, GPIO, UART. To be noted that by using platform drivers, applications and device drivers can operate without needing to know the specifics of the hardware platform they are running on. For example, in this paper is exploited the SPI platform driver module, shown in Figure 10.


SPI
spi.h
spi.cpp/.c
spi_extra.h
Platform independent SPI structures and enumerations
Platform independent SPI functions (prototypes)
Platform specific SPI functions definition
Platform specific SPI structures and enumerations

```
struct spi_desc { ... }
struct spi_init { ... }
enum spi_mode { ... }
extern spi_init() { ... }
extern spi_read_write() { ... }
extern spi_remove() { ... }

spi_init() { ... }
spi_read_write() { ... }
spi_remove() { ... }

struct mbed_spi_desc { ... }
struct mbed_spi_init { ... }
```

Figure 10: Structure of SPI platform module

D. Data acquisition from AD7606 using Nucleo-F401RE

Two approached has been followed to retrieve data from AD7606: without the no-Os driver and with the no-Os driver. The first approach is more straightforward, meaning that the code does not use the no-Os library and the commands to each GPIO has been sent referring to the timing diagram in 7. This method has been useful to understand working principles of AD7606 and the signals' evolution during conversion. SPI is not properly exploited in this case, since the clock for data retrieval is given by the pin RD which is manually driven in the code and the MISO pin is manually read from a GPIO. The pinout and wiring description is detailed in the following subsection.
Here is reported the pseudo-code representing the structure

of the code without using the no-Os driver. To be noted that AD7606 must receive a RESET pulse after power up in order to behave properly.

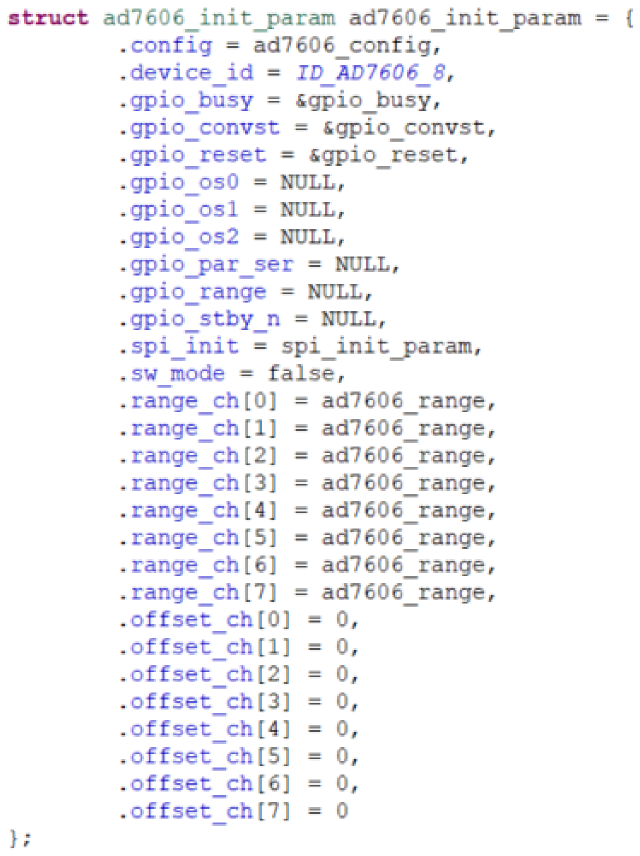
```
set CS pin LOW
set CONVST pin HIGH
set RESET pin LOW
set RESET pin HIGH
set RESET pin LOW
while 1 do
    read data
    convert 2's complement
    convert with formula in Eq. 1
    send results (HAL_UART_Transmit())
end while
```

The function to read incoming data from AD7606 is built as follows:

```
set CONVST pin LOW
set CONVST pin HIGH
set CS pin LOW
while BUSY pin is HIGH do
    wait
end while
for k=0; k<4; k++ do
    for i=15; i>=0; i– do
        set RD pin HIGH
        set RD pin LOW
        read D7 pin and shift value
        read D8 pin and shift value
    end for
    store values in buffer
end for
set CS pin HIGH
```

The second approach is more elegant since it exploits the no-Os driver already written and available online. The result is more accurate since a floating channel does not influence anymore the behaviour of its following channels, as happened in the first approach. More detailed information about this problem and other issues are in section IV.

Regarding the principles of working with no-Os drivers, in addition to configuration enumerations and structures, they provide also two key structures: the device initialization structure and the device driver structure. The device initialization structure allows users to define device-specific parameters and configurations in their application code. It contains members from other device-specific parameter structures and enumerations. In this project the device initialization structure is shown in Figure 11, where each defined GPIO is a no-Os structure containing the number of GPIO port, the number of GPIO pin and the GPIO platform.



```
struct ad7606_init_param ad7606_init_param = {
        .config = ad7606_config,
        .device_id = ID_AD7606_8,
        .gpio_busy = &gpio_busy,
        .gpio_convst = &gpio_convst,
        .gpio_reset = &gpio_reset,
        .gpio_os0 = NULL,
        .gpio_os1 = NULL,
        .gpio_os2 = NULL,
        .gpio_par_ser = NULL,
        .gpio_range = NULL,
        .gpio_stby_n = NULL,
        .spi_init = spi_init_param,
        .sw_mode = false,
        .range_ch[0] = ad7606_range,
        .range_ch[1] = ad7606_range,
        .range_ch[2] = ad7606_range,
        .range_ch[3] = ad7606_range,
        .range_ch[4] = ad7606_range,
        .range_ch[5] = ad7606_range,
        .range_ch[6] = ad7606_range,
        .range_ch[7] = ad7606_range,
        .offset_ch[0] = 0,
        .offset_ch[1] = 0,
        .offset_ch[2] = 0,
        .offset_ch[3] = 0,
        .offset_ch[4] = 0,
        .offset_ch[5] = 0,
        .offset_ch[6] = 0,
        .offset_ch[7] = 0
};
```

Figure 11: Device initialization structure

The device driver structure loads these initialization parameters through the ad7606_init() function. It is dynamically allocated at run-time from the heap. The parameters in the device driver structure are almost identical to those in the device initialization structure, effectively making the device driver structure a run-time version of the device initialization structure [9]. The initialization flow of device can be summarized:

1) Create a definition (or instance) of the device init structure: struct ad7606_init_params to initialize the user specific device parameters and platform-dependent driver parameters. The parameters are defined during compilation time.
2) create a pointer instance of the device driver structure: static struct ad7606_dev *ad7606_dev = NULL;
   This single pointer instance is used with all no-OS driver APIs/functions to access device-specific parameters. The pointer points to memory dynamically allocated in the heap.
3) Initialize the device and other platform specific peripherals by calling the device init function: int32_t ret = ad7606_init(&ad7606_dev, &ad7606_init_param).

Here is reported the pseudo-code of the overall flow from initialization till data retrieval.

```
create struct no_os_gpio_init_param gpio_busy
create struct no_os_gpio_init_param gpio_convst
create struct no_os_gpio_init_param gpio_reset
create struct stm32_spi_init_param
```

```
5:  create struct no_os_spi_init_param spi_init_param
6:  create struct ad7606_range
7:  create struct ad7606_config
8:  create struct ad7606_init_param
9:  create struct ad7606_dev *ad7606_dev = NULL
10: ad7606_init(&ad7606_dev, &ad7606_init_param)
11: create array to store data: uint32_t data[8]
12: while 1 do
13:     ad7606_read(ad7606_dev, data)
14:     convert results in data in 2's complement
15:     convert results using Eq. 1
16:     send data HAL_UART_Transmit()
17: end while
```

In this second method the already written function read() already manages the CONVST pin to start conversion and the RD pin. More precisely, the RD pin is connected to SPI_SCLK pin in Nucleo-F401RE, which automatically provides the clock. Moreover, only D7 pin is used to retrieve data and it is connected to MISO pin in Nucleo-F401RE. Hence, in this case saying that SPI protocol is used is correct.

E. Communication between Nucleo-F401RE and RaspberryPi

The communication between Nucleo-F401RE and RaspberryPi is done through UART protocol. UART (Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used for serial communication between devices. It is widely used in embedded systems to facilitate data exchange between a microcontroller and peripheral devices such as sensors, modems, or other microcontrollers. UART operates asynchronously, meaning data is transmitted without a shared clock signal between the sending and receiving devices. It transmits data serially, one bit at a time, over a single communication line, making it simple and cost-effective for long-distance communication. The data format typically includes a start bit, data bits (usually 7 or 8), an optional parity bit for error checking, and one or more stop bits. The speed of data transmission is defined by the baud rate, which specifies the number of bits transmitted per second. UART supports full-duplex communication, allowing simultaneous transmission and reception of data through separate lines for transmitting (TX) and receiving (RX). In this project the communication relies on the following parameters:

- baudrate: 115200 Bits/s;
- Data bits: 8 bits;
- Parity: none;
- Stop bits: 1.

Data sent from Nucleo-F401RE have been formatted ("ChannelNumber,Value") before being sent in order to facilitate the splitting and management by RaspberryPi.
The connection between Nucleo-F401RE and RaspberryPi is done through a USB cable.

F. Nucleo pinout and wiring

The wiring between Nucleo-F401RE and AD7606 is different in the two approaches decsribed in section III-D.
Without no-Os driver the connection is (AD7606 -> STM32):

- BUSY -> PC7
- RESET -> PC6
- CONVST -> PB15
- CHIP_SELECT -> PB14
- D7_OUT -> PC11
- D8_OUT -> PC8
- RD -> PB12

Instead, with no-Os driver the connection changes in:

- BUSY -> PC7
- RESET -> PC6
- CONVST -> PB15
- CHIP_SELECT -> PB14
- D7_OUT -> PC11 (SPI_MISO)
- RD -> PC10 (SPI_SCLK)

G. Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B (Figure 12) is a highly capable and versatile single-board computer that builds upon the success of its predecessors with several key enhancements. It features a high-performance 64-bit quad-core processor, specifically the ARM Cortex-A72 running at 1.5 GHz, which provides significant processing power for various computing tasks. This model supports dual-display output at resolutions up to 4K via its dual micro HDMI ports. It includes hardware video decode capabilities at up to 4Kp60, ensuring smooth playback of high-resolution video content. Memory options include configurations with up to 8GB of LPDDR4 RAM, enabling improved multitasking and overall system performance, particularly for memory-intensive applications. Connectivity options are extensive, with dual-band 2.4/5.0 GHz wireless LAN and Bluetooth 5.0, offering fast and reliable wireless communication. It also features Gigabit Ethernet for high-speed wired networking and USB 3.0 ports for faster data transfer rates with compatible peripherals. Additionally, the Raspberry Pi 4 Model B supports Power over Ethernet (PoE) capability via a separate PoE HAT add-on, allowing for convenient power and network connectivity over a single cable in PoE-enabled environments. For end users, the Raspberry Pi 4 Model B delivers desktop-level performance comparable to entry-level x86 PC systems. Despite these enhancements, it maintains backwards compatibility with the prior-generation Raspberry Pi 3 Model B+, ensuring continuity and ease of migration for existing projects.

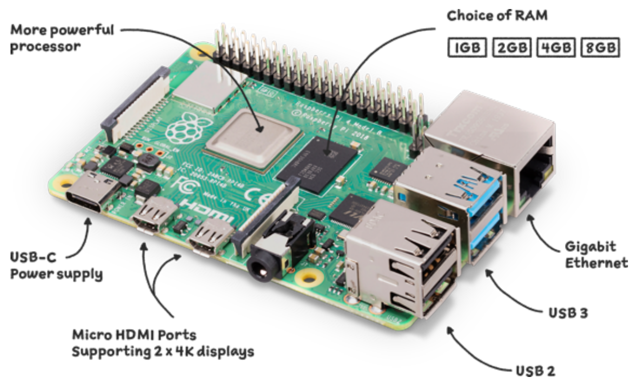


Figure 12: Raspberry Pi 4 Model B

In this project, RaspberryPi4's function is to host different services for the management of the overall IoT infrastructure. These services consists in:

- Node-RED server to manage data storage and elaboration. Data coming from USB (UART protocol) are collected thanks to a python script reading serial data and they are sent to the Node-RED server using mqtt protocol exploiting paho library.
- Python3 to run locally a script that read the serial data and send them through mqtt to Node-RED.
- InfluxDB database to store all data. It is directly connected to Node-RED server and managed by the server itself.
- Grafana server to visualize data stored in the InfluxDB database. Data is retrieved through queries written in Flux language.

To be noted that due to the limited memory of the RaspberryPi, Node-RED must be run with and additional argument to tell the underlying Node.js process to free up unused memory sooner than it would otherwise. To do this, it is necessary to use the alternative node-red-pi command and pass in the max-old-space-size argument:
node-red-pi –max-old-space-size=256 [10].

H. Management of messages from UART in RaspberryPi

In order to have a smooth management of incoming messages in RaspberryPi, a python script exploiting threads has been used. This script manages both messages from UART and messages from mqtt.
A first thread continuously listens for incoming messages from UART, which are the readings of AD7606 in all 8 channels coming from Nucleo-F401RE. Data are formatted in this way: "number of channel,value", in order to allow a perfect split by the character ",", as described in subsection III-E. Once split, the data are sent through mqtt with these rules:

- topic: number of channel
- payload: value

Mqtt configuration is built in order to provide a skelethon for future developments. It means that the structure of topics offers the opportunity to integrate further sensors and measurements to the current architecture. At the moment, the system is built as follows: ...

A second thread, instead, subscribes to mqtt topics from node-red server and manages the actions to take in each case. RaspberryPi's GPIO are used instead of re-transmitting data through UART to Nucleo-F401RE board, in order to facilitate the removal of the board if bluetooth module is required for a more efficient application.
Python module "gpiozero" has been used to activate/deactivate GPIOs in RaspberryPi to perform the required actions, like the activation of water pumps or the ventilation, according to mqtt command received. Once the action is complete, a mqtt command of success is sent back to Node-RED at the topic named "nameTopicOK" (for example PlantMonitoring/Ventilation/ManualOK") and the activation time is stored both in Node-RED dashboard in a text format, both in InfluxDB as "1" value with the exact timestamp.
The python script keeps checking for messages forever, thanks to a "while True" loop for UART and thanks to "client.loop_forever()" command for mqtt messages.

I. Node-RED

Node-RED is a programming tool for wiring together hardware devices, APIs, and online services in innovative ways. It provides a browser-based flow editor that simplifies creating and deploying flows using a wide range of nodes with a single click. JavaScript functions can be crafted within the editor using a rich text editor, and a built-in library allows for saving reusable functions, templates, and flows. The lightweight runtime, built on Node.js, leverages its event-driven, non-blocking model, making Node-RED ideal for running on low-cost hardware like the Raspberry Pi as well as in the cloud. Flows are stored in JSON format, allowing for easy import, export, and sharing via an online flow library. [11].
The architecture of the server is shown in Figure ... . Basically, the server provides an architecture able to collect data from the mqtt protocol, subscribing to the topic of interest, and subsequent nodes to elaborate these data. As first step, 8 different mqtt-in modules are inserted to collect data coming from Nucleo-F401RE board and sent by RaspberryPi python script, as described in the previous section. Each channel is then elaborated separately, according to the nature of incoming data. Let's make some examples:

- Channel 1: collects tension data from the fuel cell.
- Channel 2: collects temperature data from temperature sensor.
- Channel 3: collects humidity data from humidity sensor.

Node-red provides also a dashboard for data visualization, data selection and customization of settings. A first group in the tab called "Plant Monitoring" has been used to select the channels from which to collect data. If the switch is enabled data are collected, processed and stored in the database, otherwise all incoming data are discarded. This process is done following this flow (also in Figure 13):

1) Mqtt-in node (8 nodes, one for each channel, with topic named "1", "2", etc). All mqtt nodes refer to a Aedes mqtt broker node which runs on port 1883.
2) Function node to set the incoming value as flow variable: this variable will be useful later in the flow, if it is required to add the value to an array.
3) Switch node representing a switch in the dashboard. If it is active it lets the value pass through to be processed, otherwise the flow stops here. From now on the actions described below take place if the switch is on.
4) Function node to push the value from mqtt to a global array storing data of the corresponding channel.
5) Change node to set the value of the message payload to the flow variable containing the data.
6) Chart node to represent the trend of data in the Node-RED dashboard.
7) Function node to format data to be sent to InfuxDB.
8) Stackhero-InfluxDB-v2-write node to store data in InfluxDB.

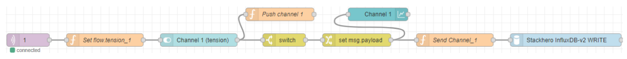


Figure 13: Node-RED flow to collect data

A second group in the dashboard tab "Plant Monitoring" allows customization of settings: it is possible to select the thresholds of values to enable further actions. For example, it is possible to set the threshold of humidity: when the collected value of humidity is below the threshold, a message (for example: "ON") is sent through mqtt to a topic (for example "PlantMonitoring/Water") and is received by RaspberryPi which subscribes to that topic. Once the RaspberryPi receives the message it does some actions, like activating a water pump to increase the level of humidity of the soil. This process has been categorized as "Automatic", since it is only necessary to set the threshold and the system is automatically monitored. To be noted that, to avoid activation of the pump too many times since the data come every few seconds and the value of humidity may not change instantaneously, a "wait" is included (i.e. if the value is under the threshold for N consecutive times, then proceed, otherwise wait). To be more precise, here is described the Node-RED flow for this process (Figure 14):

1) Text input node in dashboard to set the wanted threshold for automatic monitoring.
2) Function node (or Change node) to set the threshold as a flow variable.
3) Switch node to compare the message payload coming from the channel and the threshold. If the data value is greater than the threshold then the message is sent to the output of the node.
4) Function node to increment a counter set as flow variable.
5) Switch node to check if the counter has reached a certain number of iterations: if so, reset the counter with a change node and continue in the flow.
6) Template node to format the message to send through mqtt (for example: "ON")
7) Mqtt-out node to send the command to RaspberryPi.

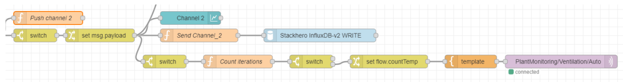


Figure 14: Node-RED flow to manage thresholds

Buttons are instead used for "Manual" commands. If pressed, the same message as in automatic mode is sent through mqtt and the same action is performed by RaspberryPi (Figure 15).

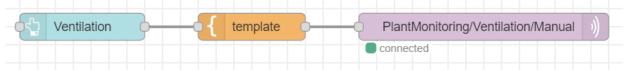


Figure 15: Node-RED flow for button commands

To be more accurate in storing data, the last timestamp of automatic and manual activation of watering or ventilation is stored in the database and visualized in the node-red dashboard for better monitoring. This is done once the response of success from the device responsible of activation of pumps is received through mqtt, in order to be sure that the process is properly done. The activation time is also store in InfluxDB database. The flow for this operation can be summarized as follows, both for manual and for automatic case (also in Figure 16):

1) Mqtt-in node for successful response (for example: "PlantMonitoring/Ventilation/ManualOK").
2) Date/time formatter node to set the correct date and time with respect to the timezone.
3) Text node to print in the dashboard the time in which the action has concluded.
4) Function node to format the data to store in the database (for example: "measurement: "Channel_2", ventManualActivation: 1, timestamp: Date.now()").
5) Stackhero-InfluxDB-v2-write node to store the activation in database.

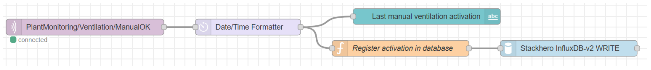

PlantMonitoring/Ventilation/ManualOK
connected
Date/Time Formatter
Last manual ventilation activation
Register activation in database
Stackhero InfluxDB-v2 WRITE

Figure 16: Node-RED flow to register activation commands

An example of the complete "Plant Monitoring" tab is shown in Figure 17.

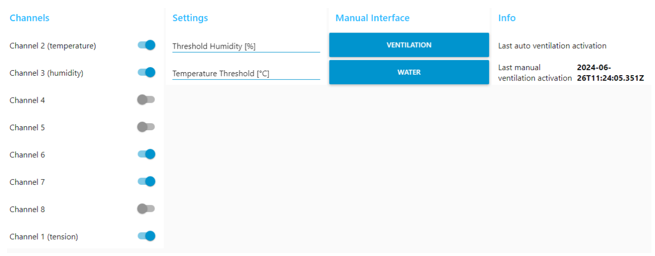
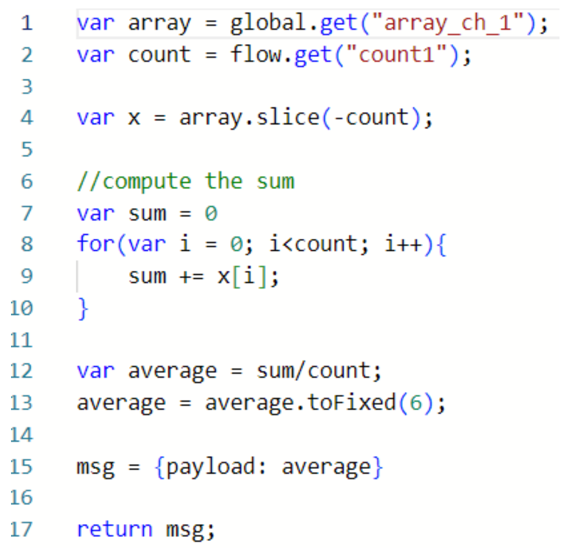


Figure 17: Example of monitoring tab in Node-RED

We then created a second tab in Node-RED dashboard called "Visualization" for quick representation of data from sensors. In this tab it is possible to monitor the trend of data thanks to a chart for each channel and a final chart collecting all channels together. The flow containing the charts has been described above.
Moreover, it is possible to monitor the mean and standard deviation of data through gauges, setting the desired number of values on which the mean or standard deviation for each channel should be computed. The flow for computing the mean is simple and can be represented as follows:

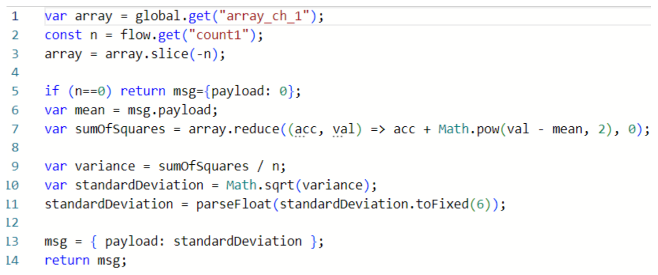
1) Numeric node to set the number of desired values on which to compute the mean.
2) Change node to set the numeric value from the previous node as a flow variable.
3) Function node to compute the mean (Figure 18) retrieving the array of values of interest from the corresponding global variable.
4) Gauge to display the value in dashboard.



```
var array = global.get("array_ch_1");
var count = flow.get("count1");

var x = array.slice(-count);

//compute the sum
var sum = 0
for(var i = 0; i<count; i++){
    sum += x[i];
}

var average = sum/count;
average = average.toFixed(6);

msg = {payload: average}

return msg;
```

Figure 18: Function for mean value computation

The flow to compute the standard deviation exploits the computation of the mean and is composed by a function node (Figure 19) and a gauge to represent data in the dashboard.



```
var array = global.get("array_ch_1");
const n = flow.get("count1");
array = array.slice(-n);

if (n==0) return msg={payload: 0};
var mean = msg.payload;
var sumOfSquares = array.reduce((acc, val) => acc + Math.pow(val - mean, 2), 0);

var variance = sumOfSquares / n;
var standardDeviation = Math.sqrt(variance);
standardDeviation = parseFloat(standardDeviation.toFixed(6));

msg = { payload: standardDeviation };
return msg;
```

Figure 19: Function for standard deviation computation

Concluding, an example of "Visualization" tab is shown in Figure 20. We inserted an additional group called "Summary" to represent all channels in one chart for a better analysis of the overall behaviour of the system.

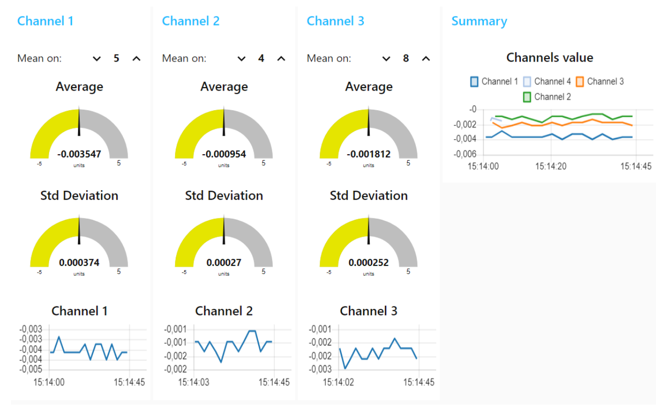


Figure 20: Example of "Visualization" tab in Node-RED dashboard

J. InfluxDB

InfluxDB is a specialized database for storing time-stamped data efficiently. It's designed to handle large amounts of data used in monitoring, analytics, and IoT applications. InfluxDB offers its own query language (InfluxQL and Flux), scales well across multiple servers, and ensures data remains accessible even during hardware failures. It integrates easily with visualization tools like Chronograf and Grafana and is available in both open-source and commercial versions to suit different needs.

In this project we used InfluxDB to store all data acquired from the sensor, sent through UART to the Raspberry and then sent through mqtt to node-red server. We created a bucket called "PlantMonitoring" that contains one measurement field for each channel and each measurement contains one or more fields, according to the nature of the channel. A basic structure is shown in Figure 21, taking as example the first 2 channels.

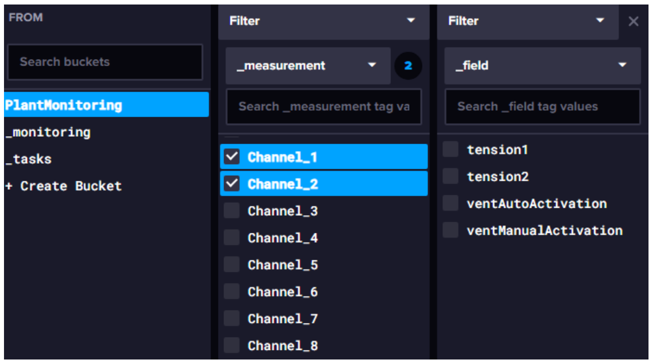


Figure 21: Example of data storage in InfluxDB

An example of data displayed in graphs is in Figure ..., (metti foto di dati reali acquisiti)

It is also possible to monitor the activation times of water pumps or other external devices, since Node-RED stores this information (as described in Section III-I), in order to create a history of activation, like in Figure 22 22.

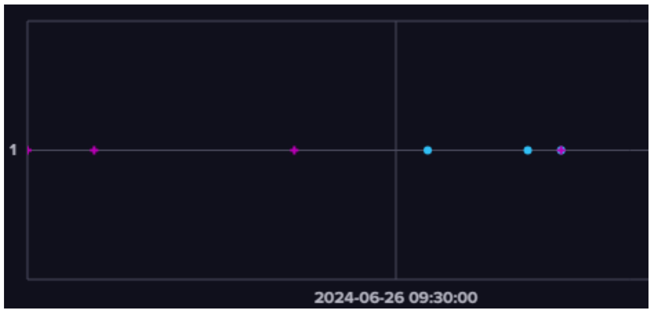


Figure 22: Example of activation time stored. Blue dots: manual mode, Purple dots: automatic mode

K. Grafana

Grafana is an open-source tool that helps you visualize and analyze metrics, logs, and data from different sources. It connects to databases and tools like InfluxDB to create graphs and dashboards that show trends and performance. You can set alerts for important events and annotate graphs with notes. Grafana also lets you create reusable templates for dashboards, making it easy to share insights across teams [12].

In this project, thanks to Grafana, we visualize the data stored in the project database, allowing us to have an immediate representation of the trend of the measurements and their history during time.
We created multiple dashboards in a playlist to represent different sets of data, which are organized as follows:

- Channel Dashboard: collects graphs of single channels values and their mean and standard deviation
- SummaryOfChannel Dashboard: represents the trend of all channels together in one plot and mean and standard deviation of all 8 channels together. This allows an immediate comparison between data coming from channels.

It is also useful to analyse the mean value and standard deviation of data. We computed the mean and standard deviation of the channels in Node-RED and displayed them in the Node-RED dashboard. However, in Grafana it is more convenient to query mean and standard deviation directly from the database. Here we report two examples of queries to retrieve mean value and standard deviation from InfluxDB database:

- Mean value: from(bucket:"PlantMonitoring") |> range(start:-1h) |> filter(fn: (r) => r._measurement == "Channel_1") |> filter(fn: (r) => r._field == "tension1") |> mean()
- Standard deviation: from(bucket:"PlantMonitoring") |> range(start:-1h) |> filter(fn: (r) => r._measurement == "Channel_1") |> filter(fn: (r) => r._field == "tension1") |> stddev()

IV. Issues and troubleshooting

This sections aims to analyse the issues encountered during the creation of the project. Firstly, the library is not as immediate to use as it seems: it requires to build precise data structures to work. For this reason a first attempt to make the AD converter work has been done using Arduino Uno and following [13]. The signals have been analysed and the code has been transcribed in C language to be used in Nucleo STM32F401RE.
V_drive should be tied to 3V3 usign STM32 since the output voltage of GPIOs is 3V3. Using Arduino, instead, V_drive can be tied to 5V.
Signals have been analysed with the oscilloscope to understand timing and working principles.
To be noted that with Arduino, the clock for SPI transfer is not the clock from Arduino itself but at each iteration, a pulse on RD pin is sent to enable the reading of data. Using STM32, instead, it is convenient to use the SPI clock provided.
With the approach without the AD7606 library, channels present an offset between them of about 3 mV as in Figure 23. This is due to the fact that the offset cannot be compensated acting on the registers of the device.

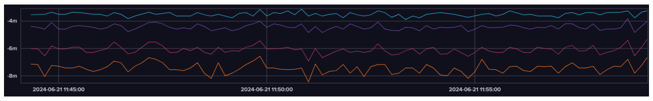


Figure 23: Offset in channel readings

Using the library, instead, the offset seems compensated, since it can be set to 0 in the code. Figure 24 shows the behaviour of the first four channels in the case that they are all set to ground. A small offset still remains in the first channel: this could be due to the fact that we used a breadboard and all channels are set on the ground line, which has some resistance.

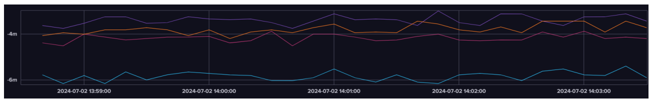


Figure 24: Data acquired from first 4 channels set to GND

V. Experimental results

To test the accuracy of the overall IoT system we built a complete infrastructure, as in Figure ... . It contains, apart from the devices already described above, the following components:

- Temperature sensor.
- Humidity sensor.
- Fans.

We collected data for some days and the result is shown ... bla bla bla

VI. Conclusion

In conclusion, this paper presented a method for continuously monitoring plant health using an AD7606 analog-to-digital converter, an STM32 Nucleo board, and a Raspberry Pi 4 as the gateway.
The data acquisition process is crucial for establishing a reliable infrastructure. To achieve the best results, the AD7606 library from Analog Devices for STM32 should be utilized. Data is transmitted via UART to the Raspberry Pi, which then forwards it to a Node-RED server for further management and visualization using InfluxDB and Grafana.
We developed a comprehensive IoT infrastructure capable of both automatic monitoring and manual control of external devices, such as water pumps and ventilation systems. Timestamps are recorded in the database to build a historical record.
Data visualization is vital for tracking plant health. In fact, this system is able to successfully monitor all plant parameters, providing a visual representation and an immediate interface to customize settings and parameters.
Lastly, further developments are feasible: the Nucleo-F401RE could be replaced with a Bluetooth Low Energy module. This would reduce the physical space needed for system assembly and enable direct data transmission to the Raspberry Pi and Node-RED.

References

[1] internet of things (iot). [Online]. Available: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT
[2] Challenges in internet of things (iot). [Online]. Available: https://www.geeksforgeeks.org/challenges-in-internet-of-things-iot/

[3] T. Tilekar, R. Tighare, M. Tidake, R. Tidke, S. Tiwadi, and P. (Cholke), "Smart plant monitoring system using iot," 03 2022.

[4] U. B. M. Anusha k, "Automatic iot based plant monitoring and watering system using raspberry pi," *International Journal of Engineering and Manufacturing(IJEM)*, no. 6, 2018. [Online]. Available: https://www.proquest.com/openview/2819b4d6f7c6c304823aa1a798133b2c/1?pq-origsite=gscholar&cbl=2069184

[5] M. H. Absar, G. F. Mirza, W. Zakai, Y. John, and N. Mansoor, "Novel iot-based plant monitoring system," *Engineering Proceedings*, vol. 32, no. 1, 2023. [Online]. Available: https://www.mdpi.com/2673-4591/32/1/12

[6] B. Suneja, A. Negi, N. Kumar, and R. Bhardwaj, "Cloud-based tomato plant growth and health monitoring system using iot," in *2022 3rd International Conference on Intelligent Engineering and Management (ICIEM)*, 2022, pp. 237–243.

[7] . R.-C. W. M. Ruscalleda Beylier, Ed., *Comprehensive Biotechnology (Third Edition)*, 2011.

[8] H. Roy, T. U. Rahman, N. Tasnim, J. Arju, M. M. Rafid, M. R. Islam, M. N. Pervez, Y. Cai, V. Naddeo, and M. S. Islam, "Microbial fuel cell construction features and application for sustainable wastewater treatment," *Membranes*, vol. 13, no. 5, 2023. [Online]. Available: https://www.mdpi.com/2077-0375/13/5/490

[9] Understanding and using the no-os and platform drivers. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/understanding-and-using-the-no-os-and-platform-drivers.html

[10] Running on raspberry pi. [Online]. Available: https://nodered.org/docs/getting-started/raspberrypi

[11] Node-red. [Online]. Available: https://nodered.org/

[12] About grafana. [Online]. Available: https://grafana.com/docs/grafana/latest/introduction/

[13] Ad7606. [Online]. Available: https://github.com/leollo98/AD7606/tree/main