Autonomous Plant Monitoring Station IoT Project Report

Giulia Titton, *240311*

***Abstract*—Monitoring systems are becoming essential in various fields, including plant health, especially with Microbial Fuel Cells (MFCs). MFCs are bio-electrochemical devices that produce electricity through bacterial metabolic activity. Ensuring the health and efficiency of these bacteria is crucial for optimal energy production. Internet of Things (IoT) systems are well-suited for this task, providing precise and continuous automatic monitoring of environmental conditions. Our IoT system includes an analog-to-digital converter (AD7606) installed near the plant station, 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 communicate with each other and the cloud, equipped with sensors and embedded systems. IoT enables data transfer over networks without the need for direct human interaction, seamlessly connecting devices such as heart monitors, agricultural biochips, and smart vehicles. These devices collect environmental data, which is then transmitted to IoT gateways for analysis by applications or back-end systems. The core components of an IoT ecosystem include web-enabled devices that use processors, sensors, and communication hardware to collect and transmit data. Connectivity is crucial, allowing these devices to communicate over internet networks and use IoT gateways as central data aggregation hubs. Data analysis aims to extract meaningful insights and address potential issues locally to optimize bandwidth consumption. Human interaction with IoT devices involves setup, instructions, and data access through graphical user interfaces like websites or mobile apps. Additionally, IoT systems often incorporate connectivity protocols, 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. 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. 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. 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 [1].

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 [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. 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.
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. Cost considerations must balance deployment expenses with long-term benefits. To overcome these challenges, we 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].

In line with this structured approach and highlighted challenges, this paper focuses on an 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, 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 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 explains how to set up the system. Section V shows some of the issues encountered during the development process of the project and their solution. Section VI presents some real data acquired while testing the system. Lastly, Section VII 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 1 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.

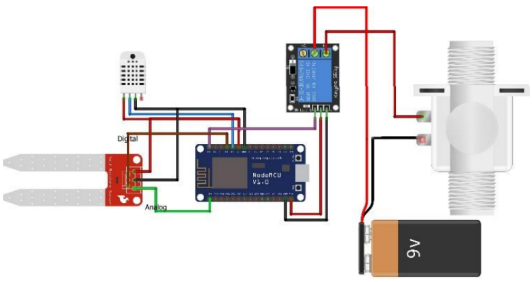

Figure 1: System proposed by [3].

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.

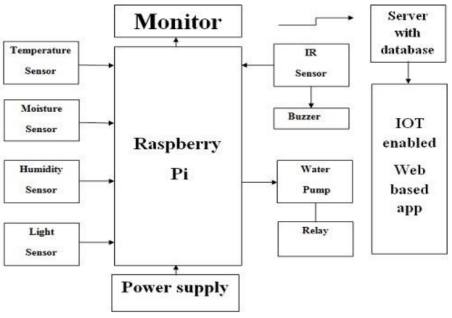

Figure 2: System proposed by [4].

A third paper [5] presents an IoT-based Smart Plant Monitoring System that uses sensors to gather environmental data for plant maintenance. The system collects temperature and humidity data from a DHT-11 sensor, processes it with an Arduino microcontroller, and transmits it to the Blynk app via an ESP8266 Wi-Fi module. The Blynk app displays real-time data and sends notifications if parameters exceed thresholds. Additionally, a solenoid valve controlled by the Arduino automates watering based on sensor readings, with

a relay managing the valve's operation. This system enables remote monitoring and control, optimizing plant environments and reducing manual intervention.
Another paper addresses agricultural challenges such as unpredictable weather and plant diseases using IoT and machine learning [6]. It employs soil moisture, humidity-temperature sensors, and a camera module for disease detection in tomato plants. The system uses an Azure Custom Vision Model for accurate disease detection and ThingSpeak for real-time data display. It includes two modules: plant health monitoring and disease detection. The former monitors air temperature, humidity, soil temperature, and moisture, adjusting watering via a relay based on sensor readings. Components include DHT11 (air temperature and humidity), DS18B20 (soil temperature), capacitive soil moisture sensor, SI114X (sunlight sensor), and a relay. This setup ensures optimal plant care and features a user-friendly GUI dashboard. Figure 3 shows the complete architecture.

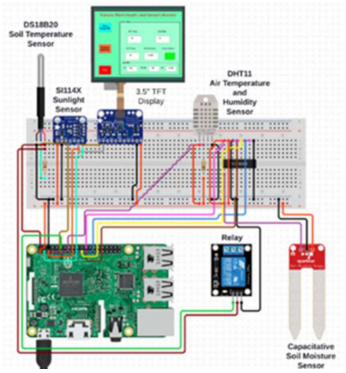

Figure 3: System proposed by [6].

As last related work, we present the most relevant to our project [7]. The author uses our same AD converter (AD7606) to acquire data and an ESP32 microcontroller to manage the IoT infrastructure. Moreover he built a custom board (PCB), shown in Figure 4, to make the whole system more efficient, self-contained and compact.

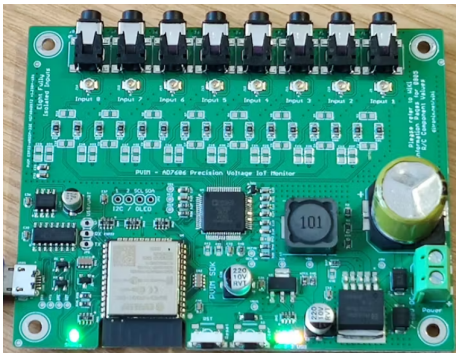

Figure 4: System proposed by [7].

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 5. Firstly, we acquire data from the MFC and convert it from an analog value to a digital value thanks to the AD7606 module. We retrieve the converted data through SPI from STM32 microcontroller on the Nucleo board which will elaborate and transfer data to Raspberry Pi employed as server. Raspberry Pi will manage the internet connectivity to the overall IoT infrastructure, based on a Node-RED server, an InfluxDB database and Grafana for data visualization. Moreover it will manage the actions to be taken according to the messages coming from Node-RED.
The next subsections will describe all the components of the system and the acquisition process from MFC to the IoT infrastructure.

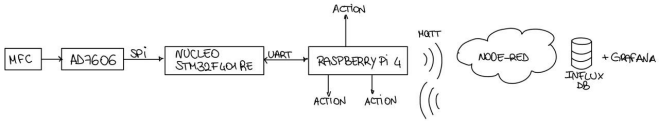

Figure 5: General working flow of our plant monitoring system.

A. Plant Microbial fuel cells (PMFCs)

Plant Microbial fuel cells (PMFCs) are systems that can generate electricity by harnessing microorganisms' metabolic activity [8]. PMFC is built using carbon brush electrodes and soil from the campus ground. It is a single-chamber fuel cell without a polymer electrolyte membrane (PEM) separating anodic and cathodic regions. Two electrodes are placed at 10 centimeters depth difference, limiting oxigen diffusion. The anode is at the bottom of the vase while the cathod is 2 centimeters below the ground. The PMFC is considered active when the tension is above 0.6V, otherwise it is considered inactive [9]. Figure 6 shows the structure of the PMFC employed.
In this study we will monitor the behaviour of PMFC by analysing tension levels and activating water pumps or ventilation systems to optimize the environment for soil bacteria.

Note that fuel cells are exploited in various scenarios in real life. For example in [10] the authors employ MFCs for Sustainable Wastewater Treatment.

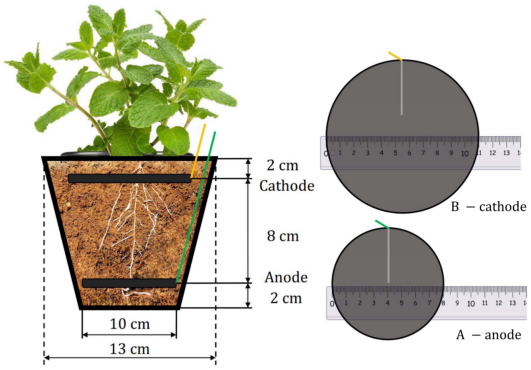

Figure 6: On the left: construction details of the reactor. On the right: dimensions of anode and cathode carbon brush electrodes, the titanium wire inserted into the electrodes [9].

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 (Figure 7). 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.

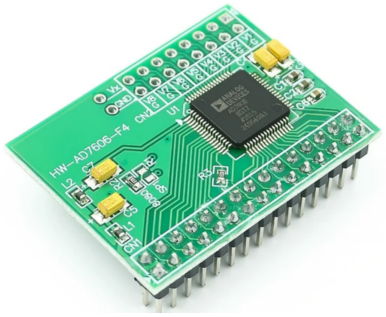

Figure 7: AD7606 evaluation board.

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:

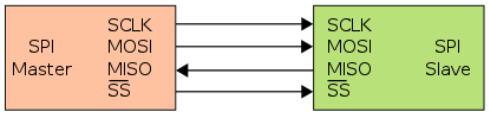

Figure 8: SPI protocol

- 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 8 shows briefly how the simplest communication between master and slave works.

In this application, the SPI protocol works as in Figure 9, 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.
It is important to note that for using the acquisition board in serial mode, resistor R1 must be connected. Without it, the device will function in parallel mode.

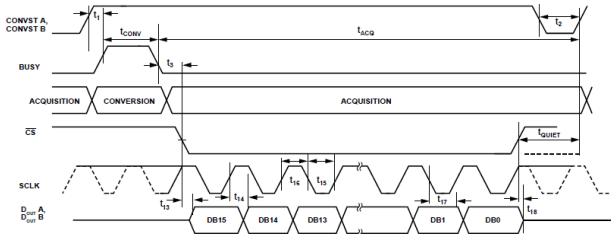

Figure 9: Serial Read Operation

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 10 and using the Equation 1. The result of conversion is the resulting level of tension.

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

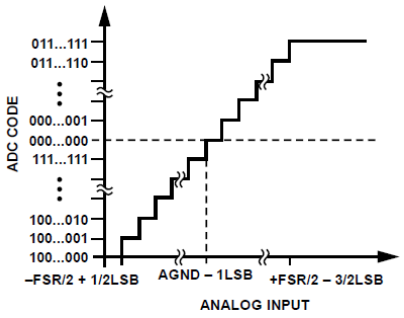

Figure 10: AD7606 Transfer Characteristic

C. Nucleo-F401RE and no-Os drivers

We used Nucleo-F401RE to manage the converted 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 [11]. 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 11.

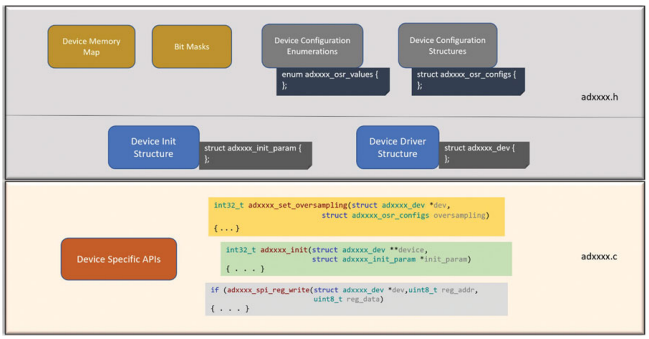

Figure 11: Structure of no-Os driver

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.

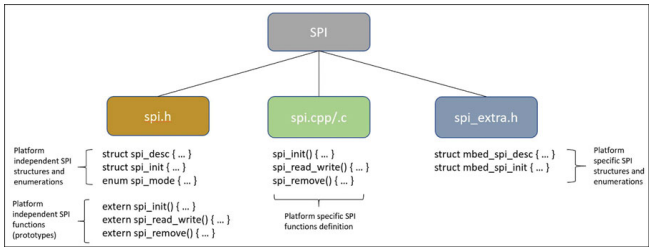

Figure 12: Structure of SPI platform module

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 12.

D. Data acquisition from AD7606 using Nucleo-F401RE

We followed two approached 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. In fact, we send commands to each GPIO considering the timing diagram in Figure 9. This method is very 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. To be noted that we checked signal evolution with an oscilloscope to be sure of the correct behaviour of AD7606.

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. Note that AD7606 must receive a RESET pulse after power up to behave properly.

1: *set CS pin LOW*
2: *set CONVST pin HIGH*
3: *set RESET pin LOW*
4: *set RESET pin HIGH*
5: *set RESET pin LOW*
6: **while** 1 **do**
7: *read data*
8: *convert 2's complement*
9: *convert with formula in Eq. 1*
10: *send results (HAL_UART_Transmit())*
11: **end while**

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

1: *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

We firstly tested this approach using Arduino Uno following the guide in [12] and then converted the code for STM32. However, we noted that floating analog channels lead to incorrect behaviour of its following channels and that we needed 2 GPIOs to read all 8 channels: D7 reads the first 4 channels and D8 reads the last 4 channels. Hence, a more precise approach is suitable.

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 previously. More detailed information about this problem and other issues are in section V.
Regarding the principles of working with no-Os drivers, in addition to configuration enumerations and structures, they provide also two key features: the device initialization and device driver structures. 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 13, where each defined GPIO is a no-Os structure containing the number of GPIO port, the number of GPIO pin and the GPIO platform. Additionally, the structure ad7606_range assigned to each channel includes details about the channel's minimum and maximum values, as well as a boolean value indicating whether the channel is differential. 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 [11]. 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;

```
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 13: Device initialization structure

   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
create struct no_os_spi_init_param spi_init_param
create struct ad7606_range
create struct ad7606_config
create struct ad7606_init_param
*create struct ad7606_dev *ad7606_dev = NULL*
ad7606_init(&ad7606_dev, &ad7606_init_param)
create array to store data: uint32_t data[8]
while 1 **do**
    ad7606_read(ad7606_dev, data)
    convert results in data in 2's complement
    convert results using Eq. 1
    send data HAL_UART_Transmit()
end while

In this second method the already written function read()

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 for SPI transfer. Moreover, only D7 pin is used to retrieve data from all 8 channels and it is connected to MISO pin in Nucleo-F401RE. Therefore, it is accurate to state that the SPI protocol is used in this instance, exploiting the 42 MHz peripheral clock provided by STM32.

E. Communication between Nucleo-F401RE and RaspberryPi

The Nucleo-F401RE and Raspberry Pi communicate using the 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.

We formatted data sent from Nucleo-F401RE as "Channel-Number,Value" to facilitate the splitting and management by RaspberryPi.
Nucleo-F401RE and RaspberryPi are connected through a USB cable.

F. Nucleo pinout and wiring

The wiring between Nucleo-F401RE and AD7606 varies in the two approaches described in section III-D. This difference is primarily because the SPI protocol is utilized only when using the no-OS driver, which requires just one data line.
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)

Note that 5V and GND pins in AD7606 are connected to 5V and GND pins in STM32 Nucleo board and VIO pin in AD7606 is tied to 5V, providing an external reference of 5V to the analog-to-digital conversion.

G. Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B (Figure 14) 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. 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.

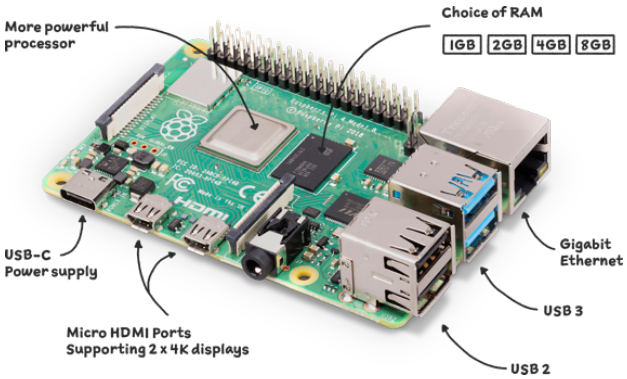

Figure 14: Raspberry Pi 4 Model B

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.
- Python3 to run locally a script that read the serial data and send them through mqtt to Node-RED.

- InfluxDB database to store received 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.

Note 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 [13].

H. Management of messages from UART in RaspberryPi

In order to have a smooth management of incoming messages in RaspberryPi, we wrote a python script exploiting threads. This script manages both messages from UART and messages from mqtt.
A primary thread continuously listens for incoming messages from UART, which consist of readings from the AD7606 on all 8 channels originating from the Nucleo-F401RE. The data is formatted as "number of channel,value" to facilitate perfect splitting by the character "," as described in subsection III-E. After splitting, the data is sent via MQTT according to the following rules:

- topic: number of channel
- payload: value

A second thread, instead, subscribes to MQTT topics from node-red server and manages the actions to take in each case. We chose to use the Raspberry Pi's GPIO rather than re-transmitting data via UART to the Nucleo-F401RE board. This approach makes it easier to remove the Nucleo board if a Bluetooth module is needed for a more efficient application or if we want to use battery-powered actuation that do not depend on energy efficiency.
We utilized the Python module "gpiozero" to control the GPIOs on the Raspberry Pi, enabling actions such as activating water pumps or ventilation according to received MQTT commands. Once an action is completed, a success MQTT command (an acknowledgment) is sent back to Node-RED on the topic named "nameTopicOK" (e.g., PlantMonitoring/Ventilation/ManualOK"). The activation time is recorded both in text format on the Node-RED dashboard and as a "1" value with the exact timestamp in InfluxDB.
The Python script continually checks for messages using a "while True" loop for UART and the "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 [14].
Key components of Node-RED are [15]:

- Nodes: Core components with embedded integration logic, deployable as npm modules.
- Editor: Visual tool for node wiring and configuration.
- Runtime: Integrates nodes for executing flows.

In this study, the server provides an architecture able to collect data from MQTT, subscribing to the topic of interest, and subsequent nodes to elaborate 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.

A more detailed description of flows will be provided later in the discussion focusing on the Node-RED dashboard.

Node-RED provides also a dashboard for data visualization, data selection and customization of settings. We created a first group in the tab called "Plant Monitoring" 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 15):

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 assign the incoming value as a flow variable, which will be useful later in the flow if there's the need to add the value to an array.
3) Switch node emulating a toggle on the dashboard. When active, it allows the value to pass through for processing; otherwise, the flow stops. Subsequent actions described below occur only when the switch is activated.
4) Function node to push the value from MQTT to a global array storing data of the corresponding channel.
5) Change node to assign 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.

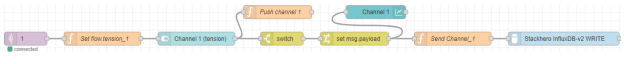

Figure 15: Node-RED flow to collect data

Then, we built a second group in the dashboard tab "Plant Monitoring" that 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 and temperature: when the collected value of humidity/temperature is below/above the threshold, a message (for example: "ON") is sent through MQTT to a topic (for example "PlantMonitoring/Water") and is received by RaspberryPi which subscribed to that topic. Upon receiving the message, the Raspberry Pi performs actions such as activating a water pump or ventilation. This process has been categorized as "Automatic", since it is only necessary to set the threshold and the system is automatically monitored.

It is important to note that to prevent frequent pump activations we implemented a "wait" mechanism. This feature can be selected with a switch in the dashboard: it can be useful if humidity or temperature data arrive every few seconds since the value may not change instantly. Specifically, if the humidity value remains below the threshold for N consecutive times, the action proceeds; otherwise, it waits. An opposite reasoning is applied for temperature: if the temperature exceeds the threshold for a certain number of iteration, then activate the fans to increase ventilation.

Clearly, if data arrive every minute or every few minutes, the delay may not be suitable since values may have already changed. The switch has been included in the dashboard to allow the user to have more freedom in monitoring the plant station.

To be more precise, here is described the Node-RED flow for this process (Figure 16):

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/less than the threshold then the message is sent to the output of the node.
4) Switch node depending on the value true/false coming from a switch in the dashboard: if true enables the iteration counting, if false sends directly the MQTT message.
5) Function node to increment a counter set as flow variable.
6) 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.
7) Template node to format the message to send through MQTT (for example: "ON")
8) MQTT-out node to send the command to RaspberryPi.

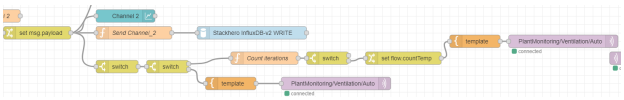

Figure 16: Node-RED flow to manage thresholds

We added then buttons in a group named "Manual Interface" to send commands manually. If pressed, the same message as in automatic mode is sent through MQTT and the same action is performed by RaspberryPi (Figure 17).

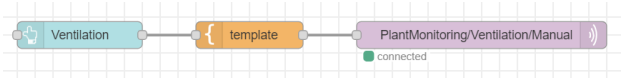

Figure 17: Node-RED flow for button commands

For enhanced data accuracy, we include an additional section titled "Info" to display the most recent times of automatic and manual watering or ventilation activations. The last activation times are stored both in the database and visualized on the Node-RED dashboard for improved monitoring. This is done once the response of success from the device responsible of activation of pumps is received through MQTT, to be sure that the process is properly carried out. The flow for this operation can be summarized as follows, both for manual and for automatic case (also in Figure 18):

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 been completed.
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.

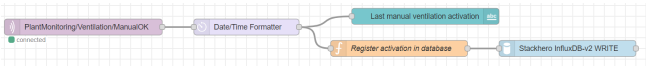

Figure 18: Node-RED flow to register activation commands

An example of the complete "Plant Monitoring" tab is shown in Figure 19.

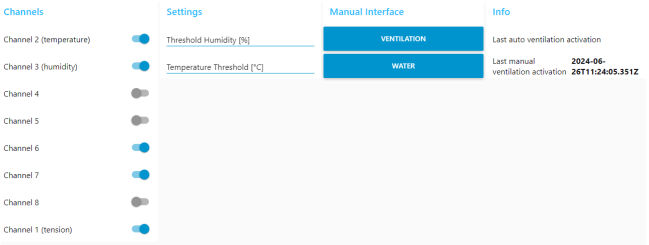

Figure 19: Example of monitoring tab in Node-RED

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

```
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 20: Function for mean value computation

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 20) retrieving the array of values of interest from the corresponding global variable.
4) Gauge to display the value in dashboard.

The flow to compute the standard deviation exploits the computation of the mean and is composed by a function node (Figure 21) 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 21: Function for standard deviation computation

Concluding, an example of "Visualization" tab is shown in Figure 22. 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.

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

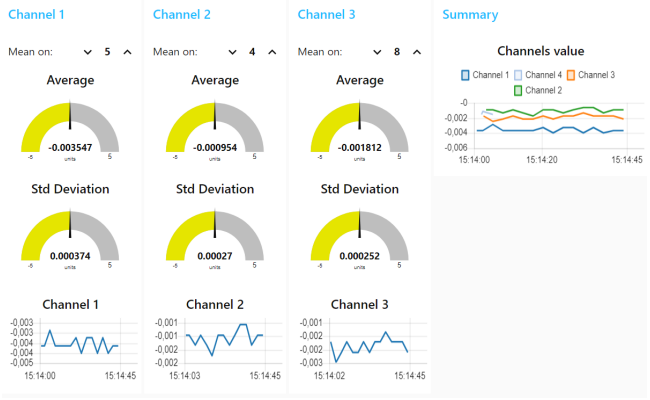

Figure 22: Example of "Visualization" tab in Node-RED dashboard

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. For example, "Channel_1" has only "tension1" as field while "Channel_2" has "tension2", "ventAutoActivation" and "ventManualActivation" since the second channel manages the temperature inputs. A basic structure is shown in Figure 23, taking as example the first 2 channels.

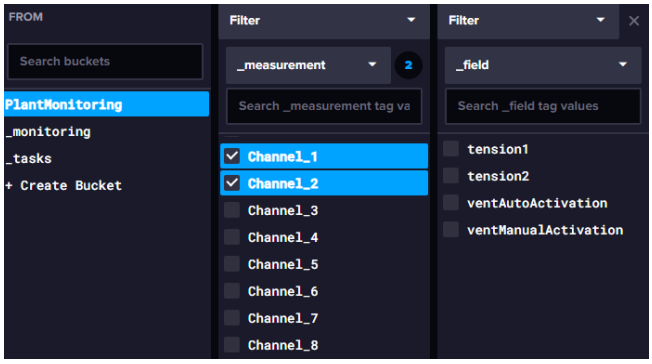

Figure 23: Example of data storage in InfluxDB

We show an example of data graph in Figure 24. It shows the first 4 channels, which are set to ground.

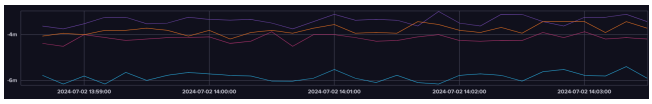

Figure 24: Data acquired from first 4 channels set to GND

It is also possible to monitor the activation times of water pumps or fans, since Node-RED stores this information (as

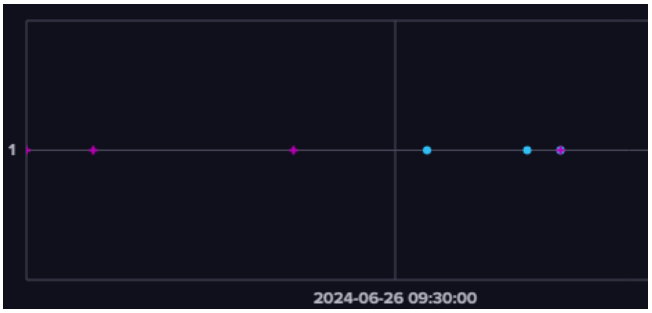

Figure 25: Example of activation time stored. Blue dots: manual mode, Purple dots: automatic mode.

described in Section III-I), in order to create a history of activation, like in Figure 25.

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 [16].
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. This allows an immediate comparison between data coming from channels.
- ActivationRecords Dashboard: shows in tables the activation times of water pumps and ventilation.

To retrieve data from buckets we used the query: *from(bucket:"PlantMonitoring")* |> *range(start:-1h)* |> *filter(fn: (r) => r._measurement == "Channel_1")* |> *filter(fn: (r) => r._field == "tension1").*
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, since the computation considers all the elements stored. 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()

Figure 26: Example of the first and second channels in "ChannelDashboard".

Figure 27: Example of the third and second fourth channels in "ChannelDashboard".

Examples of the first dashboard ("ChannelDashboard") is shown in Figure 26 and Figure 27. Each row of the dashboard represents a channel, with its own mean and standard deviation to allow better visualization and analysis.

The second dashboard ("SummaryOfChannels") collects all channels together to allow comparison between behaviour of all 8 channels. For better visualization channels are split in 2 groups of 4 channels. The query used to retrieve all data and build a graph is more complex:

import "influxdata/influxdb/v1"
data = from(bucket: "PlantMonitoring") |> *range(start: -1h)*
|> *filter(fn: (r) => r._measurement == "Channel_1" or r._measurement == "Channel_2" or r._measurement == "Channel_3" or r._measurement == "Channel_4")*
|> *filter(fn: (r) => r._field == "tension1" or r._field == "tension2" or r._field == "tension3" or r._field == "tension4")*
data = from(bucket: "PlantMonitoring")
|> *range(start: -1h)*
|> *filter(fn: (r) => r._measurement == "Channel_1" or r._measurement == "Channel_2" or r._measurement == "Channel_3" or r._measurement == "Channel_4")*
|> *filter(fn: (r) => r._field == "tension1" or r._field == "tension2" or r._field == "tension3" or r._field == "tension4")*
source1 = data
|> *filter(fn: (r) => r._measurement == "Channel_1" and r._field == "tension1")*
|> *yield(name: "Channel 1")*
(same code for the other three data sources)

Figure 28: Example of "SummaryOfChannels" Dashboard.

Figure 28 represents an example of "SummaryOfChannels" dashboard, where the first 4 channels are set to ground while the last 4 channels are floating.

"ActivationRecords" dashboard is very useful to monitor activation times of water pumps and ventilation both in automatic and manual mode. Data is presented in a table showing time in format: "YYYY-MM-DD HH:MM:SS". The query used to retrieve data is for example: *from(bucket:"PlantMonitoring") |> range(start:-3d) |> filter(fn: (r) => r._measurement == "Channel_3") |> filter(fn: (r) => r._field == "waterAutoActivation").*
Figure 29 represents an example of tables in the dashboard.

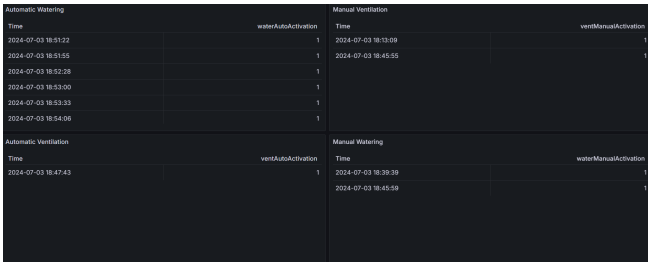

Figure 29: ActivationRecords dashboard in Grafana.

IV. How to set up the system

This section explains how to set up the system to make it work in an infrastructure. the following steps must be followed:

1) Power on Raspberry Pi.
2) Connect Raspberry Pi to a Wi-Fi network.
3) Connect through SSH to Raspberry Pi (for simplicity): ssh pi@raspberrypi.local, password: raspberry.
4) Navigate in Raspberry Pi to /home/pi/Desktop.
5) Run the script to acquire and manage data: python3 manage_pi.py.
6) In a new tab navigate to /home/pi/Desktop/grafana_v11/bin/ and run ./grafana-server.
7) Access Node-RED: http://<IPaddr>:1880. Node-RED dashboard is at http://<IPaddr>:1880/ui.
8) Access InfluxDB: http://<IPaddr>:8086.
9) Access Grafana: http://<IPaddr>:3000.

Remember to enable channels of interest in Node-RED dashboard, otherwise data are not acquired and stored!

V. 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 the project in [12]. The signals have been analysed with the oscilloscope to understand timing and working principles and the code has been transcribed in C language to be used in Nucleo STM32F401RE as described in Section III-D.
Note that with Arduino code, 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 in SPI_SCLK pin.
Using the first approach without the AD7606 library, channels present an offset between them of about 3 mV as in Figure 30. This is due to the fact that the offset cannot be compensated acting on the device registers.

Figure 30: Offset in channel readings

Using the no-Os library, the offset appears to be compensated, as it can be set to 0 in the code. Figure 31 illustrates the behavior of the first four channels when they are all set to ground. A minor offset remains in the first channel, likely because we used a breadboard where all channels share a common ground line with some resistance.

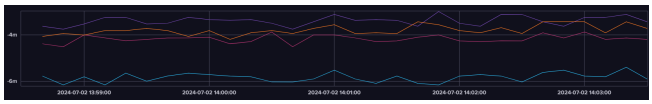

Figure 31: Data acquired from first 4 channels set to GND

VI. Experimental results

We conducted several experiments, and the results were highly unexpected. Initially, we discovered that the AD7606 evaluation board is incapable of performing differential or pseudo-differential measurements as stated in the datasheet. This conclusion was drawn from an experiment where we connected the fuel cell to the first channel of the AD7606. Instead of obtaining a voltage measurement, the device began charging the fuel cell, which is both unacceptable and unforeseen behavior; we couldn't measure the voltage between the anode and cathode due to the lack of a reference point.
We also found that all channels share a common ground, making differential measurements impossible.

On the other hand, single-ended measurements proved to be viable. We tested the system by connecting a 1.5V battery to the first channel and monitored the battery voltage. The measurement was accurate, registering 1.491V with variations

only below the millivolt range, confirming the feasibility of single-ended measurements (Figure 32).

Figure 32: Battery tension measure

Even if a differential measure is not possible, we believe that the system could be anyway exploited using single-ended measures. In fact, we additionally thought about a complete application to test and run properly all the IoT system features including:

- Temperature sensor.
- Humidity sensor.
- Fans.
- Water pumps.
- Led.
- Relays for actuation.

Figure 33 shows also the circuit for actuation.

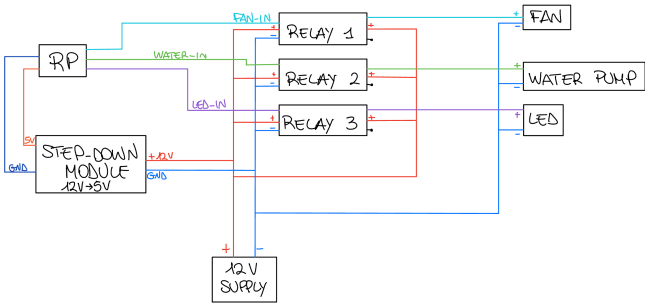

Figure 33: Complete actuation system.

VII. 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 parameters, providing a visual representation and an immediate interface to customize settings and parameters.

A drawback is the non-differential nature of the AD7606, despite the datasheet claiming it to be differential. Therefore, the MFC's voltage cannot be directly monitored. However, the overall system can still be used effectively.

Lastly, further developments are feasible: the connection between Nucleo-F401RE and Raspberry Pi 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. Furthermore, optimizing code in Node-RED is crucial for enhancing performance and minimizing memory usage. Currently, the code is duplicated across all channels, even though they execute identical actions initially.

Additionally, it would be beneficial to develop a method for measuring differential data from the MFC, as we currently lack a reference for these measurements.

With these enhancements, this project promises to deliver reliable, efficient, and precise monitoring of a plant station, empowered by seamless remote access capabilities.

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] Pvim esp32 ad7606 precision voltage iot monitor sdk board. [Online]. Available: https://www.hackster.io/DitroniX/pvim-esp32-ad7606-precision-voltage-iot-monitor-sdk-

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

[9] M. Doglioni, M. Nardello, and D. Brunelli, "Plant microbial fuel cells: Energy sources and biosensors for battery-free smart agriculture," *IEEE Transactions on AgriFood Electronics*, vol. PP, pp. 1–11, 01 2024.

[10] 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

[11] 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
[12] Ad7606. [Online]. Available: https://github.com/leollo98/AD7606/tree/main
[13] Running on raspberry pi. [Online]. Available: https://nodered.org/docs/getting-started/raspberrypi
[14] Node-red. [Online]. Available: https://nodered.org/
[15] Low code microservices with node-red. [Online]. Available: https://medium.com/engineered-publicis-sapient/low-code-microservices-with-node-red-cef1a5b4d852
[16] About grafana. [Online]. Available: https://grafana.com/docs/grafana/latest/introduction/