Created in LABVIEW 2021

21

“Programmer’s Guide”

Course: Measurements for Automation and Industrial Production

Prepared by:

ANTONIO PEDICINI (389000080)
MIRIAM PARADISO (389000052)
HAMZA MEKNI (389000063)
OMID MORADIARSHAD (389000067)

Table of contents

Abstract

"The project aims to develop an automated measurement system for characterizing the Renesas FS-1012 analog flow sensor transducer.

The task involved calibrating the raw voltage measurements of an analog sensor using multiple hardware instrumentation.

During the project, the LabVIEW environment was used to

- *develop virtual instruments to control a power supply and*
- *an Arduino counterpart that generates a voltage source and acquires airflow measurements produced by a digital flow sensor, respectively.*

At the same time, a virtual instrument dedicated to collecting airflow measurements from the analog flow sensor was created.

On top of that, a signal conditioning circuit was built to amplify the raw voltage sampled from the analog sensor, whose amplitude was too small to be used.

The airflow was produced by an air pump, connected in series with the airflow sensors. The project concluded with the calibration of the analog sensor based on the measurements taken with the system so built".

Chapter I

Programmable Hardware Instrumentation Used

Power Supply Agilent E36436A

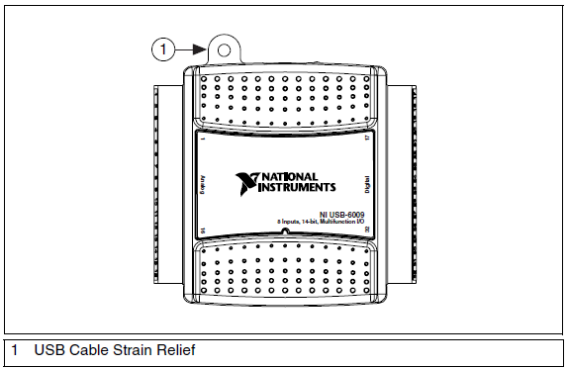
The Agilent E3633A/E3634A DC Power Supply was configured for Constant Voltage (CV) operation, with the output voltage fixed at 12V.

Figure 1: Power Supply E3634A

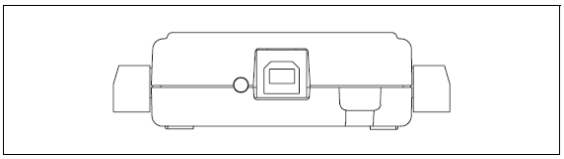
Main characteristics

- **Voltage:** $\pm(\%$ of setting + offset)
  - **Low Range (0 to 8 V):** $\pm(0.05\% + 5$ mV)
  - **High Range (0 to 20 V):** $\pm(0.05\% + 15$ mV)
- **Current:** $\pm(\%$ of setting + offset)
  - **Low Range (0 to 20 A):** $\pm(0.2\% + 5$ mA)
  - **High Range (0 to 10 A):** $\pm(0.2\% + 5$ mA)

NI DAQ USB-6008 data acquisition



(a) USB-6008



(b) USB-6008 back view

Figure 2

USB6008 Datasheet[1]

The NI USB-6008 provides connection to eight analog input (AI) channels, two analog output (AO) channels, 12 digital input/output (DIO) channels, and a 32-bit counter with a full-speed USB interface.

The following block diagram shows key functional components of the USB-6008:

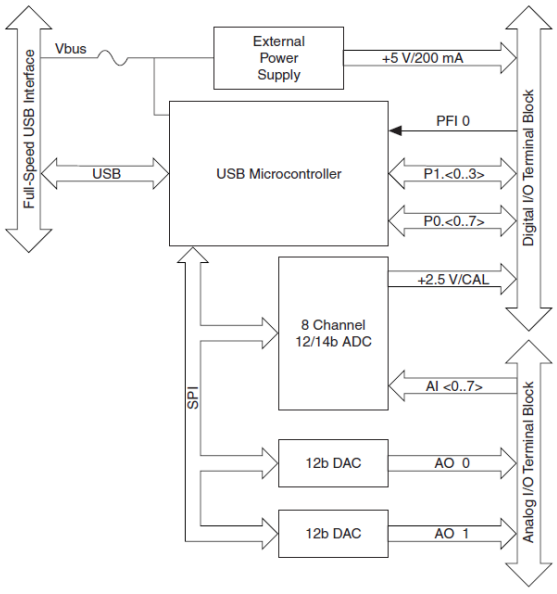


Figure 3

[1] https://www.ni.com/docs/en-US/bundle/usb-6008-specs/page/specs.html?srsltid=AfmBOoomX5uQUUn1iMT0b3gol5ElTAT813--B6ktlUhKk4vixIqbCJly

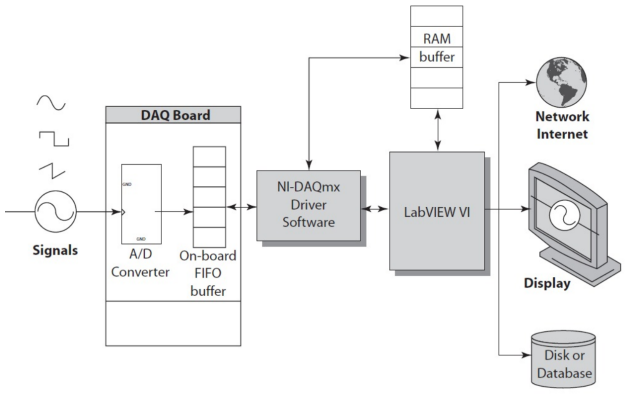


Figure 4

The capabilities of the NI USB-6008 include:

- **Analog Input:** Acquisition of analog signals from external sensors or devices.
- **Analog Output:** Generation of analog signals for driving actuators or simulating processes.
- **Digital I/O:** Reading and writing of TTL digital signals, crucial for control and logic applications.
- **Counter/Timers:** Counting of digital events using a high-resolution 32-bit counter.
- **Power Sourcing:** Provision of regulated +5V and +2.5V outputs to power external low-power devices.

Internal Architecture

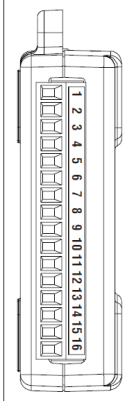
The device incorporates:

- A 12-bit ADC that, through a multiplexer (MUX) and a programmable gain amplifier (PGA), allows to selection and acquisition of one of the analog channels in differential or single-ended mode;
- two 12-bit DACs to generate 0–5 V analog signals;
- an integrated USB microcontroller, which manages data transfer, timing and control logic;
- an internal FIFO for data buffering during acquisition.

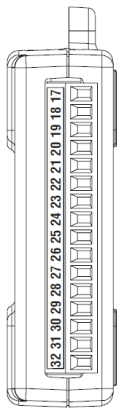
All channels are accessible via removable screw terminals, divided between analog and digital terminals, which allow a quick and flexible connection to signal cables (from 16 to 28 AWG).

The device supports:

- Analog acquisition modes:
  - Single-ended (RSE): all channels measure with respect to GND.
  - Differential (DIFF): measurements on channel pairs, with greater noise immunity.
- External digital trigger (via PFI 0) for measurement synchronization with external events.
- Event counter for counting logic transitions (up to 5 MHz).
- Terminal Assignments

Module	Terminal	Signal, Single-Ended Mode	Signal, Differential Mode
	1	GND	GND
	2	AI 0	AI 0+
	3	AI 4	AI 0–
	4	GND	GND
	5	AI 1	AI 1+
	6	AI 5	AI 1–
	7	GND	GND
	8	AI 2	AI 2+
	9	AI 6	AI 2–
	10	GND	GND
	11	AI 3	AI 3+
	12	AI 7	AI 3–
	13	GND	GND
	14	AO 0	AO 0
	15	AO 1	AO 1
	16	GND	GND

(a) Analog Terminal Assignments

Module	Terminal	Signal
	17	P0.0
	18	P0.1
	19	P0.2
	20	P0.3
	21	P0.4
	22	P0.5
	23	P0 6
	24	P0.7
	25	P1.0
	26	P1.1
	27	P1.2
	28	P1.3
	29	PFI 0
	30	+2.5 V
	31	+5 V
	32	GND

(b) Digital Terminal Assignments

Figure 5

Arduino UNO R3

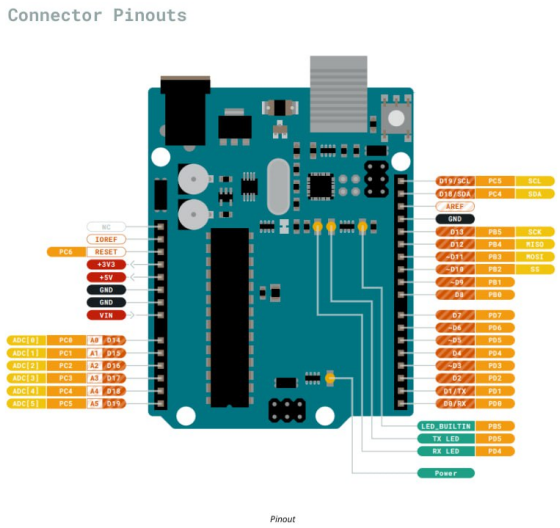


Figure 6: Arduino UNO R3 Pins

Arduino is an open-source platform encompassing a programmable electronic board and an Integrated Development Environment (IDE). The IDE facilitates writing, compiling, and uploading C/C++ based programs (sketches) to the board, leveraging simplified custom libraries and functions like **digitalWrite()** and **analogRead()**, along with a Serial Monitor for real-time communication. A typical Arduino Uno board integrates a central **Microcontroller Unit (MCU)** (e.g., ATmega328P) with built-in memory (Flash, SRAM, EEPROM) to execute code and manage I/O. Its diverse pins include **Digital Input/Output (I/O) Pins** (0–13) for digital control, **PWM Pins** (e.g., 3, 5, 6, 9, 10, 11) for analog simulation, and **Analog Input Pins** (A0–A5) connected to a 10-bit Analog-to-Digital Converter (ADC) for reading variable voltages (0–1023). Essential **Power Pins** (5V, 3.3V, GND, VIN) provide regulated power, while a **USB Port** enables PC connection for code upload, power, and serial communication. A **Reset Button** restarts the MCU, and a **Voltage Regulator** ensures stable internal power.

Chapter II

Conditioning Circuit

Circuit Design

The Conditioning Circuit was specifically engineered to amplify the output signals from the analog sensor under calibration. These signals are inherently low-amplitude, rendering them unsuitable for precise detection of airflow variations within the sensor. The sensor provides two distinct outputs: a voltage generated by thermopile TP2, exposed to unheated airflow, and a voltage from thermopile TP1, subjected to a temperature gradient. The latter exhibits a voltage variation (approximately 15mV at maximum input flow range, as per the datasheet) induced by a strategically placed resistor between the two thermopiles. Upon heating, this resistor elevates the temperature of the airflow directed towards the sensor's output, where thermopile TP1 is situated. Consequently, an electrical potential component, attributable to the Seebeck effect, causes a voltage deviation from the no-flow condition.

Nevertheless, experimental measurements using a multimeter corroborate the sensor's datasheet specifications, indicating that the voltages from both thermopiles are in the range of hundreds of millivolts.

Furthermore, to mitigate the risk of overheating the current-controlled air pump, we elected to operate significantly below the sensor's maximum input range, limiting the flow to a maximum of 6 L/min (compared to a specified maximum of 10 L/min). This operational constraint implies that the observed voltage variation will be substantially less than the 15mV indicated in the datasheet.

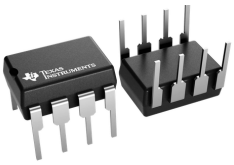
In essence, the available signals are insufficient to accurately discriminate airflow passage and thus obtain reliable measurements.

Given these limitations, the implementation of a differential amplifier became necessary.

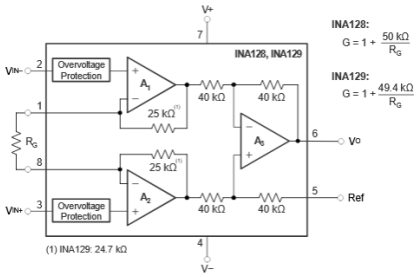
INA128P

Fundamentally, the raw signals available are inadequate for reliably discriminating airflow and thus obtaining accurate measurements.

Therefore, the integration of a differential amplifier was deemed essential. We specifically selected the Texas Instruments INA128P differential amplifier.



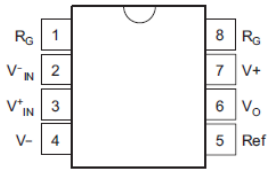
(a)



(b) Simplified Schematic

Figure 7: INA128P package (a) and its corresponding simplified schematic (b).

The INA128 is a low-power, precision instrumentation amplifier, ideally suited for measurement applications demanding high accuracy, minimal drift, and excellent common-mode rejection. Its integrated three-operational amplifier design facilitates a wide range of configurable gains via a single external resistor, thereby ensuring exceptional versatility. Key performance characteristics contributing to the device's efficiency include a low offset voltage (Max 50 μV), low quiescent current consumption (700 μA), and an outstanding Common Mode Rejection Ratio (CMRR).



(a) Pin Configuration

PIN		TYPE	DESCRIPTION
NAME	NO.		
REF	5	Input	Reference input. This pin must be driven by low impedance or connected to ground.
R_G	1,8	—	Gain setting pin. For gains greater than 1, place a gain resistor between pin 1 and pin 8.
V–	4	Power	Negative supply
V+	7	Power	Positive supply
$V_{IN–}$	2	Input	Negative input
V_{IN+}	3	Input	Positive input
V_O	6	Output	Output

(b) Pin Functions

Figure 8: Pin INA128P

Regarding pin configuration, according to the information reported in the instrument's datasheet in Figure 8, terminals *1* and *8* are designated for connecting the gain resistor; terminal *7* receives the

power supply signal; the output signal is drawn from terminal *6*; terminals *4* and *5* are connected to a common ground; and terminals *3* and *2* are connected to the signals from which the difference is to be taken.

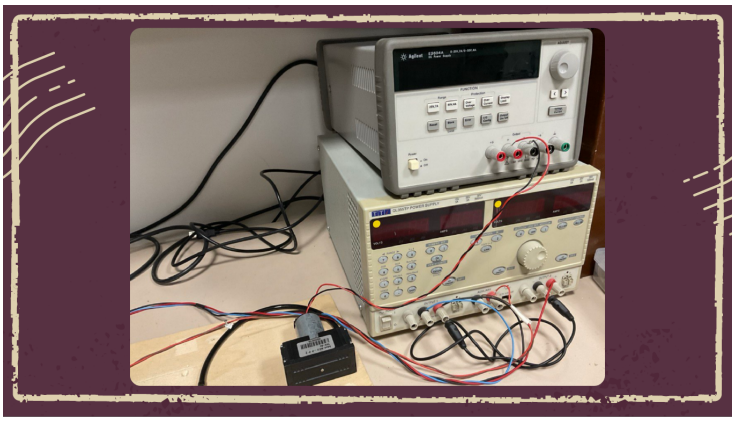
The gain factor is calculated using the formula provided in the datasheet, where the amplifier's internal feedback resistance is fixed (50 kΩ), while the gain resistance can be varied by connecting an appropriately valued resistor to terminals 1 and 8. During the experimental phases, the gain resistance was set using a potentiometer to the desired value, according to the gain table in the instrument's datasheet.

Potentiometer

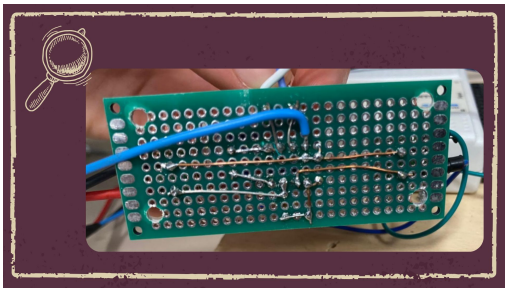
The circuit incorporates a Bourns 3296W-1-103LF multi-turn precision potentiometer, employed for fine-tuning the gain of the differential amplifier. The potentiometer is configured as a resistive divider, with its central terminal (wiper) connected to the amplifier's feedback node. Rotating the trimmer clockwise increases the resistance between the wiper and the CCW terminal, thereby adjusting the gain based on the input signal conditions. Through the potentiometer, the gain resistance was set to 100 Ω, with a tolerance of $\pm 10\%$ as per the device's datasheet. This configuration corresponds to a gain factor of 500. Consequently, if the initial difference between the two signals varies by a few millivolts, the resulting amplified signal will vary between a few hundred of millivolts. The potentiometer was connected to the pins *1* and *8* of the INA128P.

Devices Interconnections

The conditioning circuit is connected to the measurement system instruments via jumper wires. These interconnections are fully explained in the ***User's Manual***. Hence, please refer to that manuscript for more information. In this section, simple illustrations are reported for reference.

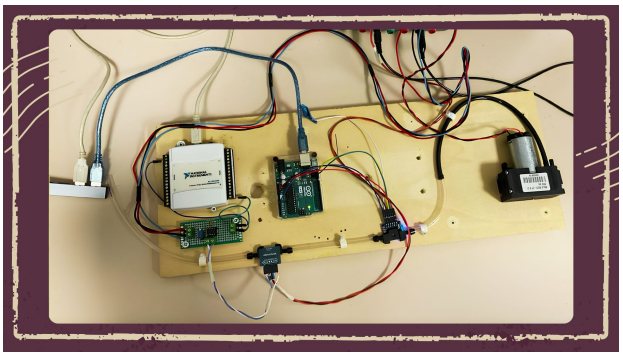


(a) Power Supplies: Agilent E3634A & QL355TP

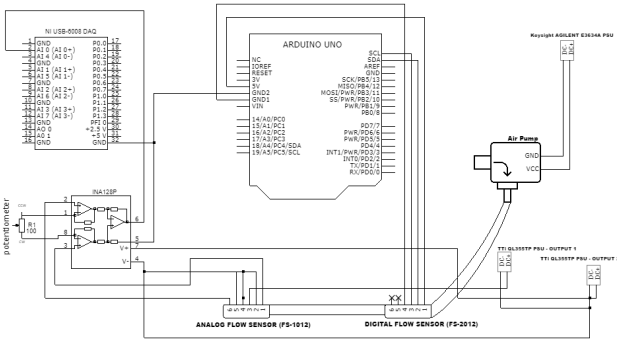


(b) Backside of the Conditioning Circuit Printed Board

Figure 9: Experimental setup components: Power supplies and the backside of the conditioning circuit board.



(a) Physical Measuring System Setup



(b) Schematic of the Whole Measuring System

Figure 10: Overview of the measuring system: Physical setup (left) and detailed schematic (right).

Chapter III

Software

LabVIEW Virtual Intrument

The study of the components of the measurement system, along with the delicate task of managing the acquisition of measurement samples simultaneously from multiple devices through different instrument interfaces, led to the choice of the **LabVIEW** development environment. Graphical programming in LabVIEW greatly facilitates the programmer's task, thanks to the wide selection of libraries and software extensions, which offer a high-level abstraction compared to low-level programming of interfaces such as **GPIB** or the **Serial** interface. Furthermore, LabVIEW is a programming environment that was specifically designed to handle the acquisition and generation of input and output signals, respectively, through **DAQ** data acquisition boards.

For these reasons, in order to develop a software system capable of managing the various instruments present in the physical measurement system, we developed several applications (or ***Virtual Instruments (VI)***), each tasked with handling communication with a given device through different programming interfaces.

Figure 11: Icons representing different components (VIs) of the system.

Before any modifications can be made to the program, the reader should be familiar with the concepts of *Enum*, *Queue Operations*, *Event Case Structure*, and *Finite State Machines* as implemented in LabVIEW. Therefore, a brief overview of some of the software paradigms used in this project will be provided, with particular focus on Finite State Machines (FSM) and the Producer-Consumer design pattern.

Finite State Machine (FSM) LabVIEW Design Pattern

A **Finite State Machine** (or FSM) is a computational model used to represent a system that can be in only one state at a time, and that transitions from one state to another in response to specific events or conditions. It is one of the most commonly used paradigms for designing control logic, operational sequences, and reactive systems.

In LabVIEW, a finite state machine can be implemented in various ways. For this project, the FSM has been implemented using a queue-based state scheduling mechanism, made possible through the **Synchronization/Queue Operations** *sub-palette*. Figure 12 shows the architecture adopted for all FSMs within this project. Each FSM consists of a **while loop** that iteratively executes the machine, a **Guard Clause**—a case structure that handles potential errors during execution—and a second case structure responsible for decoding and executing the current state's actions, as well as managing the scheduling of the next state.

Outside the while loop, a queue is initialized and used to post state transition commands. The queue's data type is defined as an **Enum** constant, which maps integer values to string labels, thereby abstracting the machine's states into human-readable representations.

The subVIs implementing this FSM paradigm are **Agilent_GPIB_FSM.vi** and **Arduino_Serial_FSM.vi**, which control the logical sequencing and operations required to remotely manage communication with the **Arduino UNO** board and the **Agilent e3634A** power supply, respectively.

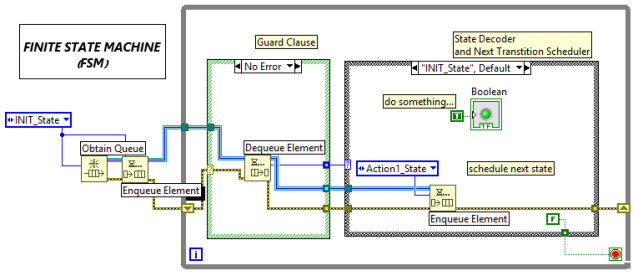


Figure 12: Finite State Machine (FSM) Example Implementation in LabVIEW

Producer-Consumer LabVIEW Design Pattern

The second paradigm employed, upon which both **main.vi** and the subVI **Arduino_FlowPump_Controller.vi** are based, is the **Producer-Consumer (Event-Driven)** design pattern. This approach is particularly well-suited for real-time applications that must respond promptly to user-generated events through the graphical user interface (GUI), while simultaneously ensuring that the program remains non-blocking—that is, each event is handled asynchronously and independently from others.

This architecture consists of an event-handling structure, referred to as the 'producer', which monitors all relevant user-GUI interactions based on the software's functional requirements. The producer defines the behavioral response for each incoming event, but is unaware of the actual implementation details necessary for event handling. These are delegated to a separate structure known as the 'consumer'. Once the consumer receives a command from the producer, it executes the corresponding operations necessary to complete the event-handling cycle.

Figure 13 illustrates the block diagram implementing a finite state machine alongside the event-based Producer-Consumer pattern. The events correspond to user-GUI interactions. Each relevant interaction triggers a state transition within the finite state machine, which is implemented in the consumer loop using an enumerated constant to manage state flow.

In this project, a multi-consumer architecture was adopted, where each consumer is designed to respond to events of a different "nature"—that is, events that require access to different software or hardware resources.

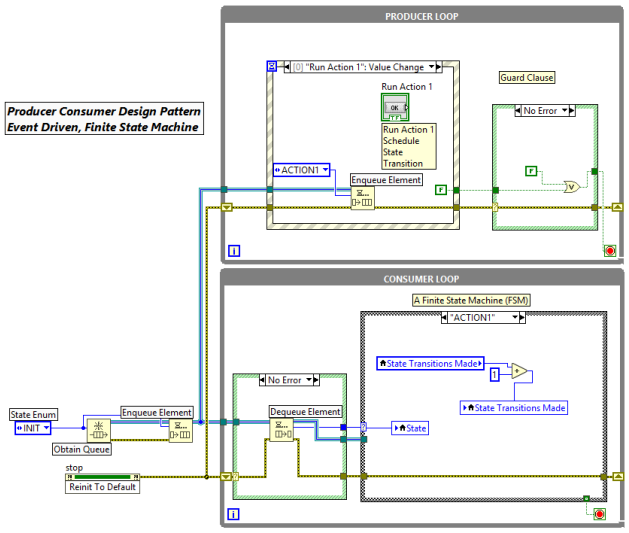


Figure 13: Producer Consumer Design Pattern Example Implementation

main.vi

The Virtual Instrument (VI) used for calibrating the analog sensor and with which the user primarily interacts is named ***main.vi***. This VI does not directly manage the connection with the measurement instruments in the system; rather, it orchestrates a set of subroutines (or **subVIs**), shown in Fig. 11, which in turn, through the libraries available in LabVIEW, acquire measurement samples and convert the control commands addressed to the instruments into the correct syntax for the respective communication interface.

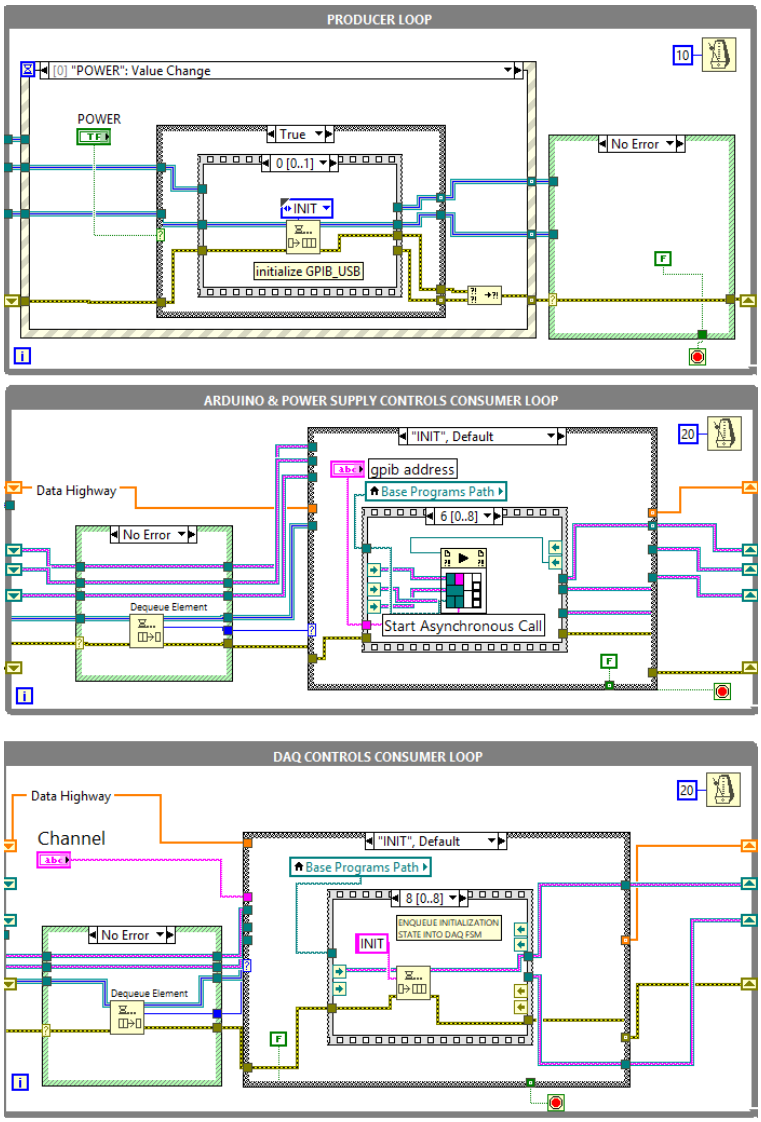
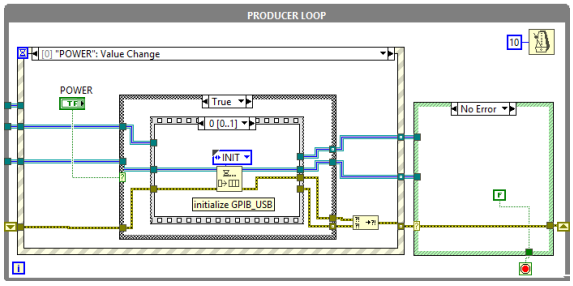


Figure 14: *main.vi Producer-Consumer Queued State Machine*

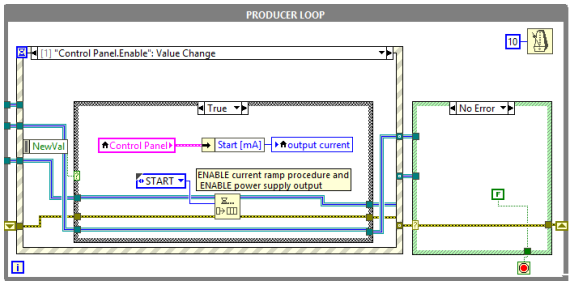
Specifically, two distinct types of consumer structures were implemented: one to manage the events that trigger state changes in the devices controlled via the Serial and GPIB interfaces—namely the Arduino UNO board and the power supply unit—and a second consumer dedicated to handling the NI DAQ-6008 data acquisition device, which samples the voltage measurements from the analog sensor to be calibrated. Each of these event handlers internally implements an additional architectural paradigm—namely, the finite state machine based on queued state scheduling (Queued State Machine). In general, finite state machines are used to model the behavior of systems in which, for a given input, the output depends on the system's current state. In the context of this project, states are used to model the connection status between the main program (LabVIEW) and the devices controlled via

their respective interfaces. For example, a given input, such as reading flow measurements from the digital sensor, will only produce valid data if the connection to the Arduino board has been successfully initialized. Similarly, a request to output a specific current from the power supply to drive the air pump can only be fulfilled if the requested current does not exceed the limits configured on the instrument. Both of these constraints—the validity of the Arduino UNO connection and the upper current limit—define a state for their respective devices.

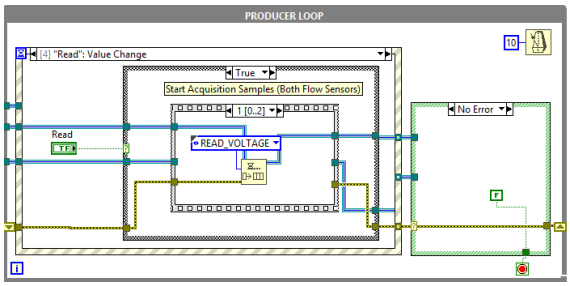
Producer Loop



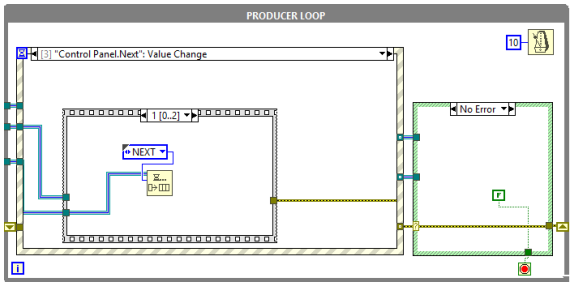
(a) When the main.vi is run the Producer Schedules the Initialization State in each Consumer Loop



(b) When *Enable* is pressed on the GUI, the Current Ramp is activated in the Start state



(c) When *Read* is pressed on the GUI, the signal acquisition from both sensors is activated



(d) When *Next* is pressed output current changes by one step up (or down, depending on the state)

Figure 15: A collection of four event handling cases, demonstrating how the Produecer schedules state transitions for the Consumer loops that run in parallel.

The internal structure of the producer loop includes an event-handling structure where the programmer defines a set of cases to respond to specific events. Notification from the producer to the consumer loops about an event is achieved through the use of queues. It is important to note that the producer is placed in parallel with the consumers, specifically stacked vertically above them, which is essential for the correct execution of the program. Indeed, if the loops were placed side by side, due to the way

graphical programming in LabVIEW is designed, the consumer loops would not be correctly executed. This is shown in Figure 14. The queues used in the producer-consumer communication ensure that all events are properly received and processed by the consumer loops. Four different event cases are shown in Figure 15.

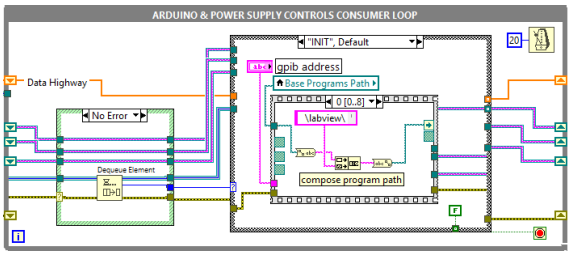
The first case is activated at the start of the main.vi execution, to command the two consumer loops to enter intialization states. Figure 15a shows the use of the first consumer loop **queue reference** and the **INIT** constant Enum value used to represent the state transition. This is all inserted in a stacked sequence to minimize space occupied. The second frame would show the same state scheduling, this time for the second loop, which handels the DAQ connection.

The second, third and fourth cases are vaery similar and also related to each other, in the sense that they usually executes in order since these cases refer to the push buttons the user interacts with to control the **Current Ramp** used to generate an ascending and descending current values input sequence to the air-pump to generate various air-flows.

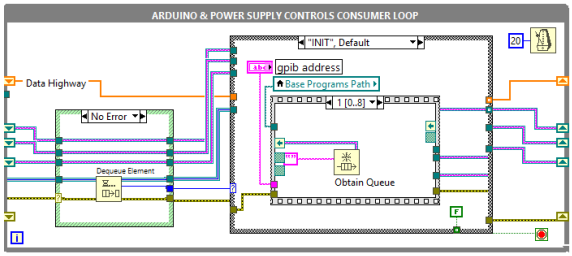
Further, to handle potential errors during program execution, each producer/consumer structure includes a **Guard Clause**—a case structure that changes color (green or red) depending on whether the connected terminal contains an error-free cluster or a cluster with an error code. When the error cluster contains an error (identified by the corresponding alphanumeric code), the border of the structure turns red. In this case, an error signal is sent to the consumer loops, which safely terminate their execution, releasing any resources previously acquired. If no error is present, execution continues with the Producer loop returning to its listening state, while the consumer loops proceed with their respective state transitions.

Finally, notice that, since the producer loop must run fast enough to respond promptly to GUI-driven user events, an iteration delay of 10 ms was introduced using the **Wait for Next Multiple ms** function, with a constant value of 10.

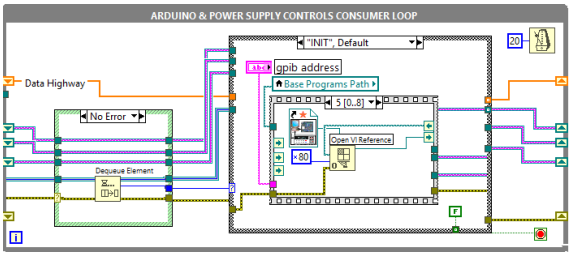
Arduino & Power Supply Controls Consumer Loop



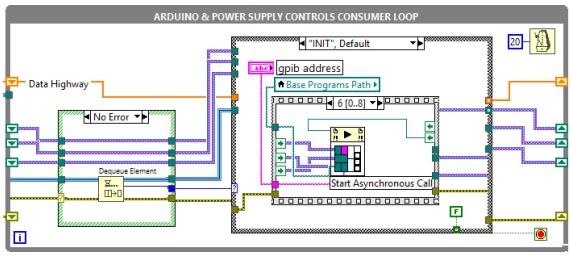
(a) Arduino & Power Supply Controls Consumer Loop: INIT state, frame 0



(b) Arduino & Power Supply Controls Consumer Loop: INIT state, frame 1



(c) Arduino & Power Supply Controls Consumer Loop: INIT state, frame 5



(d) Arduino & Power Supply Controls Consumer Loop: INIT state, frame 6

Figure 16: A collection of four frames from the sequence of operations that execute in the INIT state within the first consumer loop. The loop initializes the necessary controls to start the subVI **Arduino_FlowPump_Controller.vi** asynchronously

To handle user-GUI interactions related to the control of the Arduino UNO board and the power supply for the air pump—communicated respectively via the Serial and GPIB interfaces—a single consumer loop was implemented. Although this design choice may seem counterintuitive, given that the two communication channels are independent of each other, it was necessary because both connections are managed by a dedicated subVI named **Arduino_FlowPump_Controller.vi**. This subVI, in turn, calls two other subVIs, **Agilent_GPIB_FSM.vi** and **Arduino_Serial_FSM.vi**, which control the GPIB and Serial interfaces separately, establishing a hierarchical structure.

Asynchronous Call

By default, subVIs in LabVIEW are executed synchronously. In this mode, the calling VI halts its execution while the subVI performs its assigned task—defined by its API—and resumes only when the subVI returns control, potentially along with output data. This execution model adheres to the principle of

information hiding, which promotes encapsulation and modularity by abstracting implementation details. However, synchronous execution forces the caller to wait, which may be undesirable in systems requiring responsiveness or concurrent processing.

In contrast, an asynchronous call allows the calling VI to continue executing independently of the subVI. This feature is particularly useful for launching background processes—such as remote instrument control via Serial or GPIB—without blocking the main VI. User-GUI interactions are meanwhile collected into a queue; once the asynchronous subVI is launched, each interaction is handled in the order it was received, with no disruption to the main application. From the caller's perspective, the subVI's internal execution remains entirely opaque and self-contained.

In our system, asynchronous calling was employed to launch the subVI **Arduino_FlowPump_Controller.vi**, which manages the interface to the Arduino UNO board and the power supply unit. This subVI is started from within the consumer loop that handles instrument communication.

Figure 16 provides an overview of the main components involved in setting up the asynchronous call:

- **(a)** shows the generation of the *file path* that is used for pointing to the memory location on the user workstation to **Arduino_FlowPump_Controller.vi**.
- **(b)** illustrates the creation of one of the three queues used for: (1) sending control commands, (2) transferring sensor samples acquired by the Arduino, and (3) communicating the current setpoints for the power supply.
- **(c)** presents the use of the **Open VI Reference** function, which prepares the VI reference required for launching the subVI.
- **(d)** shows the invocation of the **Start Asynchronous Call** function, which executes the subVI in a non-blocking fashion.

The asynchronous mechanism is based on two core functions from the *Application Control* palette:

1. **Open VI Reference.vi**: This subVI generates a reference to the target VI to be executed asynchronously. Three terminals must be configured:
    - **type specifier VI Refnum (for type only)** – connected to a **Strictly Typed Static VI Reference**, which is a prerequisite for asynchronous invocation.

- **application path** – a *path* control pointing to the file location of the subVI.
- **options** – a hexadecimal constant that specifies the execution mode. Setting this terminal to **x80** indicates a **Call and Forget** mode, meaning the subVI is launched without collecting any output.

2. **Start Asynchronous Call.vi**: This subVI starts the asynchronous execution of the VI reference generated above. When configured correctly, it displays the subVI's input/output terminals, allowing parameter wiring. It also returns a reference to the launched subVI, which can be retained by the caller in case termination or interaction is required later.

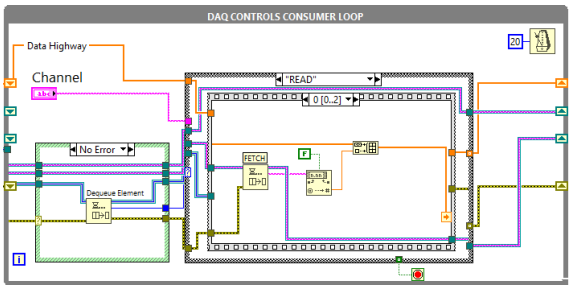
In our application, the strictly typed reference to **Arduino_FlowPump_Controller.vi** is connected to its respective inputs: the GPIB address of the power supply unit, the path to the resources folder, and three queues passed by reference. These queues serve the following functions:

- **Command Queue**: sends control commands to manage the connection states.
- **Sensor Data Queue**: transfers the digital sensor data sampled by the Arduino, ensuring no samples are lost (*lossless communication*).
- **Current Setpoint Queue**: passes the real-time current output value in mA generated the power supply.

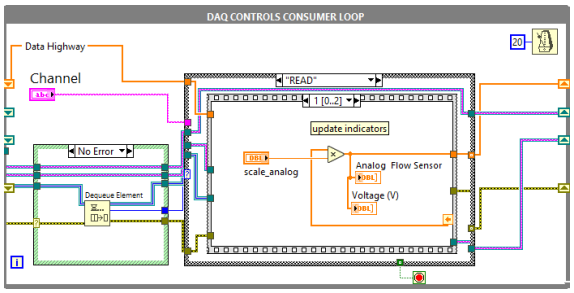
This configuration allows the subVI to run concurrently in the background while the main VI remains responsive to further user interaction and other tasks.

DAQ Consumer Loop

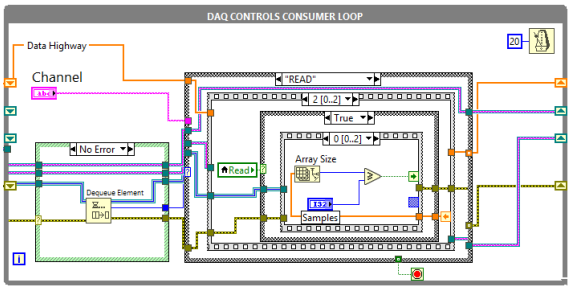
The second type of consumer loop, which handles data acquisition using the NI DAQ-6008 board, implements a pattern identical to that described for Consumer 1. Upon initialization of main.vi, a dedicated queue is created, and its reference is subsequently used by the producer loop to control state transitions exclusively for events related to data acquisition via the DAQ device. As previously described, the states are represented individually by numerical constants.



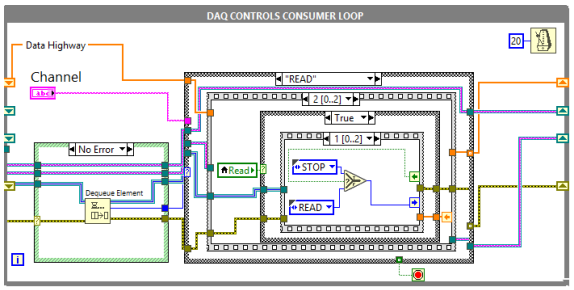
(a) DAQ Controls Consumer Loop: READ state, frame 0



(b) DAQ Controls Consumer Loop: READ state, frame 1



(c) DAQ Controls Consumer Loop: READ state, frame 2 (I)



(d) DAQ Controls Consumer Loop: READ state, frame 2 (II)

Figure 17: A collection of the frames that executes in sequence when the DAQ consumer loop enters the READ state. This happens whenever the user requests to acquire new samples during a calibration procedure run, by pressing the READ button on the front panel.

Among the few states that model the connection with the DAQ device, the one that deserves more attention is the READ state. Whenever the user requests a new data acquisition from both sensors by pressing the *READ button* on the front panel, the Producer schedules a **read state** into both consumer loops.

Figure 17 provides an overview of the main components involved in reading data from the measurement queue and defining the next state transition:

- **(a)** shows how the consumer loop fetches a new voltage sample acquired and transferred by the DAQ FSM subVI presented later. The sample is first **dequeued** by the READ state and then converted into a double numeric value. Finally, the sample is concatenated with previous samples already drawn out of the queue (if any).

- **(b)** illustrates the updating process of the indicators shown on the front panel. Samples are multiplied by a given factor to match the desired measurement unit requested by the user.

- **(c)** shows the comparison between the requested number of samples to be acquired (specified on the front panel by a numerical control that the user modifies before acquisition starts) and the current number of samples drawn from the measurement queue.

- **(d)** result of the previous comparison controls the next state transition. Whenever enough samples are acquired, the next state transition is set to be into the STOP state, which stops acquisition by clearing the **DAQmx Task**. Instead, if the number of samples is less than the desired value, the next state will still be the READ state.

Notice that the READ state executes and reschedules itself until the number of samples acquired matches the desired value expressed by the user on the front panel. Nevertheless, the user can stop the READ state in the middle of the acquisition process by deactivating the *READ button* on the front panel, shown in Figure 18. Whenever that happens, the Producer enqueues the STOP state into the second consumer loop state queue, hence stopping the acquisition procedure.

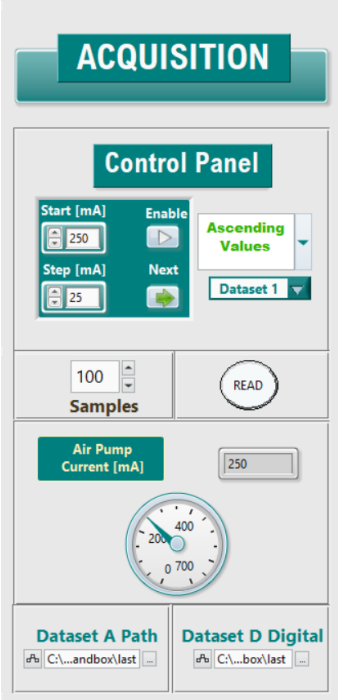
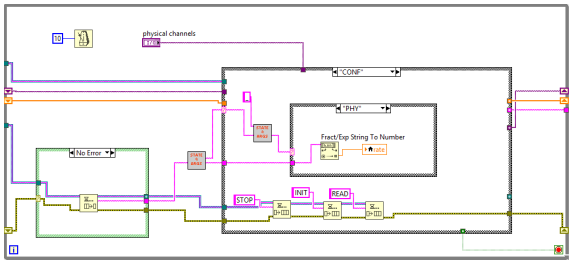


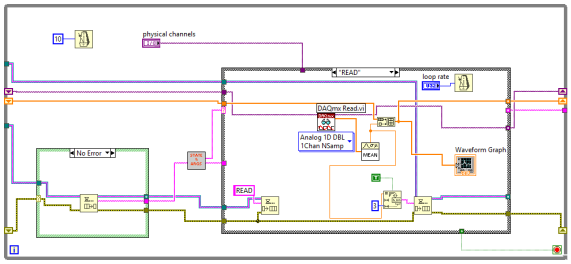
Figure 18: Acquisition Tab in main.vi shows the **Control panel** and the many interactive controls such as the *READ* button, the *Next button* and the *Samples control*.

daq_fsm.vi

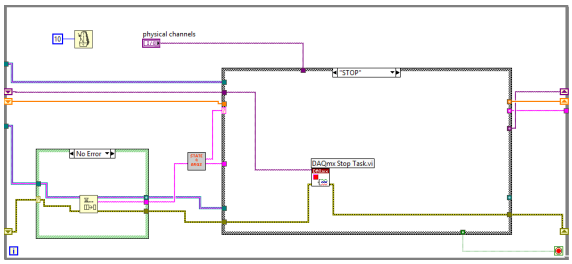
The subVI daq_fsm.vi is designed to manage the interconnection with the data acquisition (DAQ) board, which allows sampling of the output signal from the conditioning circuit. The latter, described in its dedicated section in this manual, receives as input—among other things—the two output signals from the analog sensor, that is, a pair of voltages generated by the thermopiles composing the sensor. The conditioning circuit then computes the difference between these two voltages and amplifies the result by a gain factor. The resulting signal is then connected to one of the pins of the DAQ board to be sampled and digitized.



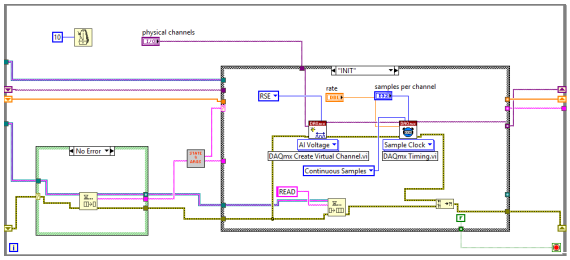
(a) daq_fsm.vi: CONF state



(b) daq_fsm.vi: ASYNC state



(c) daq_fsm.vi: STOP state



(d) daq_fsm.vi: INIT state

Figure 19: A collection of the frames that executes in sequence when the DAQ consumer loop enters the READ state. This happens whenever the user requests to acquire new samples during a calibration procedure run, by pressing the READ button on the front panel.

The DAQ board connection, via USB, leverages *National Instruments (NI)* LabVIEW drivers to facilitate communication with acquisition hardware. These drivers provide libraries for signal generation and acquisition on *virtual channels*, which are abstractions of physical I/O points managed by *Tasks*.

Fundamentally, **daq_fsm.vi** implements a finite state machine (FSM) that models the lifecycle of a virtual channel. Its default, initialization state (Fig. 19d) creates and configures the virtual channel

for analog voltage input (*AI:Voltage*), specifically as *RSE* (Referenced Single-Ended), where the measurement system shares a common ground. This state also sets default acquisition parameters via the *DAQmx Timing* task, defining sample count, continuous acquisition mode, and sampling frequency. The initialization state then transitions to the *READ* state.

In the *READ* state (Fig. 19b), the *DAQmx Read.vi* automatically commences acquisition, using the pre-configured parameters to sample the input signal. "Samples per channel" defines the acquisition board's internal buffer size, while "rate" specifies the DAQ sampling frequency. With *Continuous Acquisition*, data is continuously buffered, overwriting older data in the circular queue if space is insufficient.

Crucially, the *loop rate* control determines the repetition frequency of the READ state's 'while' loop. This timing directly impacts the program's ability to read data swiftly enough to prevent buffer saturation and data overwrite.

From the READ state, three transitions are possible: to the *STOP* state (Fig. 19c) to halt the task, back to the initial initialization state, or back to the *READ* state itself.

Arduino_FlowPump_Controller.vi 

The subVI Ardunio_FlowPump_Controller.vi, which is part of the GPIB_UART.lvlib library. This subVI controls the connection with the Arduino UNO and the programmable Power Supply Unit (PSU)- which in turn controls the air-pump- via the *Serial* and *GPIB* interfaces, respectively. As before, the presence of separate consumer loops stems from the need to simultaneously manage two fundamentally different types of connections, each of which must be controlled independently without interfering with the other. These two connections refer to the GPIB interface, used to control the Agilent e3634A power supply for driving the air pump, and the UART serial interface, employed to manage the acquisition of measurements from the digital sensor via the Arduino UNO board. Hence, the subVI calls two other subVIs that handle these connections: **Arduino_Serial_FSM.vi** and **Agilent_GPIB_FSM.vi**.

As in the previous case, the producer loop is structured as a while loop containing an event structure, followed by a guard clause designed to handle potential malfunctions either at the current level or in the lower layers (i.e., the consumer loops governed by state transition programming).

Since each event that triggers the producer loop is associated with a descriptive label, the reader

is referred to the block diagram documentation for further details on how each event is handled and which consumer loops are involved.

Arduino_Serial_FSM.vi

The firmware on the Arduino UNO's *ATmega328P* microcontroller is responsible for acquiring, converting, and transmitting flow measurements from the calibrated digital sensor. This read-convert-transfer cycle initiates upon the first successful PC-Arduino connection, which requires prior configuration of the serial port and baud rate. Once active, the serial connection allows the user to read and visualize real-time airflow data. The sampling frequency can also be modified dynamically, provided the connection is active. Users can terminate and re-establish connections to adjust parameters like serial port or baud rate.

These interaction requirements are modeled as states within a finite state machine (FSM), implemented by the **Arduino_Serial_FSM.vi** subVI from the **GPIB_UART.lvlib** library. The *Serial Connection Consumer Loop* asynchronously initiates this subVI and manages state transitions by sending high-level commands via a dedicated queue whenever the user requests a connection state modification. This abstraction, depicted in Figure 20, means the loop operates without directly engaging low-level functions like VISA.

(a) CHANGE_BAUD state. (b) CONNECTION state.

Figure 20: Collection of two states in the Serial Connection FSM consumer loop in **Arduino_FlowPump_Controller.vi** that handles the connection with the Arduino UNO board, which samples the measurements from the calibrated digital sensor.

Separating the logic that handles state transitions from the implementation of the operations performed within each respective state promotes a clear distinction between high-level and low-level program layers, fostering modularity and independence between them.

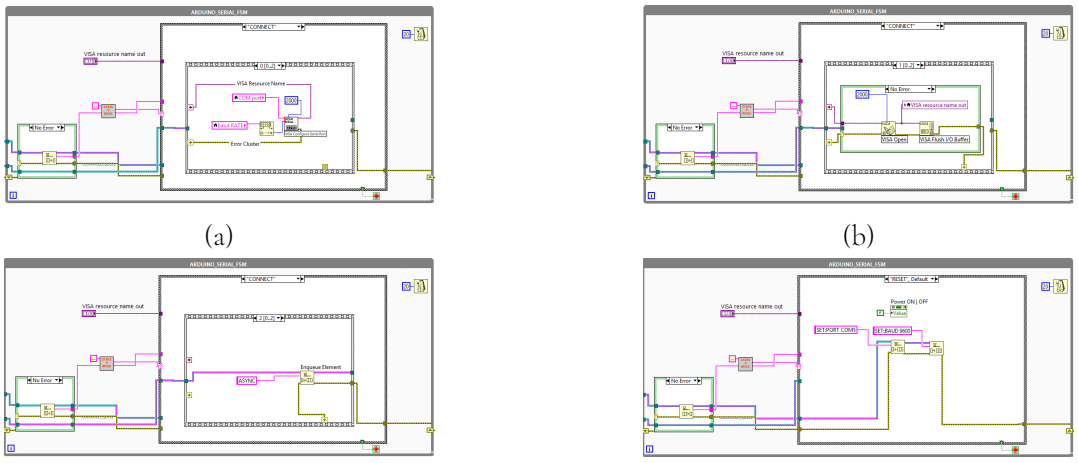


Figure 21: Arduino_Serial_FSM.vi Block Diagram.

The subVI named **Arduino_Serial_FSM** implements the most elementary form of a finite state machine (FSM) within the graphical programming paradigm of LabVIEW. To function properly, the subVI requires two queues: one for the data read from the Arduino board and another for the commands issued by the calling subVI. Additionally, an error cluster must be connected to ensure proper error handling.

The key difference between this FSM implementation and the one used in the consumer loops previously described lies in the representation of the finite automaton's states. Here, the states are not represented by customized *enum* constants, but rather by alphanumeric strings. This allows each state to be expressed as a string of characters. Furthermore, each state—interpreted as a command to be executed—can be associated with an input, or argument, that enriches the informational content of the state transition being executed.

The drawback of this approach is that states are not as easily selectable as in the case of custom controls, where the next state can be chosen from a drop-down menu. Moreover, using strings introduces the possibility of typographical errors during programming, potentially leading to transitions toward non-existent states (e.g., intending to transition to state "READ", but instead typing "RED"). This risk can be mitigated by defining a "safe" default state—one that does not produce any unintended effects regardless of the current state. In the case of the subVI described here, this default state corresponds to the *"RESET"* state, as illustrated in Figure 21d. Although this state reinitializes the resources and parameters used for the connection, it does not alter the current connection, since these modifications

do not affect the active resources and parameters but only those that might eventually be used for a new connection after the current one is terminated.

(a) State_and_args.vi SubVI Description (b) State_and_args.vi SubVI Block Diagram

Figure 22: State and Args subVI used to decode the next state transition in the Arduino_Serial_FSM.vi

In the queue used to control state transitions in the finite state machine implemented by the subVI managing the connection to the Arduino board, strings are inserted in the following format:

COMMAND (delimiter) *Argument*

To decode the content of these strings, a dedicated subVI has been developed. This subVI accepts two input strings: the string to be decoded and the delimiter that separates the state (i.e., the command) from its arguments (i.e., input parameters). It returns two output strings, corresponding to the parsed command and the recognized parameters.

The connection to the Arduino board via the serial interface is established using the parameters selected by the user, which are used to configure the *VISA Configure Serial Port* function, as shown in Figure 21a. If the port is successfully initialized, a new session is opened using the *Open Session* function, which returns a unique identifier for the established connection. Since the next state is automatically programmed to perform asynchronous reading from the microcontroller, the memory buffer is cleared to ensure that the program will read only the most recently acquired data.

By asynchronous reading, we refer to the process of reading from a memory area without blocking program execution until data becomes available from the producer (i.e., the Arduino board). In this implementation, although the current state re-enqueues itself in the state transition queue, a different state may still be scheduled by the calling subVI, namely *Arduino_FlowPump_Controller.vi*. If this occurs, since the queue operates under a FIFO (First-In-First-Out) policy, the current state will be executed once more before proceeding to the newly scheduled one (e.g., *"DISCONNECT"*). The subsequent state

might not automatically reprogram the asynchronous reading state, in which case the continuous reading from the buffer may be interrupted. In the ASYNC state, shown in Figure 23, the program uses the **VISA Read** function from the *Instrument I/O / VISA* subpalette to read data from the serial connection. Then, it enqueues the data acquired (one sample at a time) in the queue used by the higher-level subVI Arduino_FlowPump_Controller, specifically by the first consumer loop.

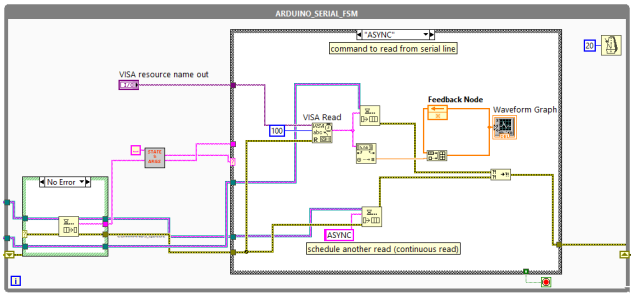


Figure 23: Asynchronous Read state used to read measurements asynchronously from Arduino board

Agilent_GPIB_FSM.vi

The connection used to control the power supply unit (PSU) driving the air pump is established via the GPIB interface. Consequently, the consumer loop handling events that affect the state of the remotely controlled instrument must use commands defined by the protocol, and specifically, by the PSU's programming manual. The table below lists the GPIB commands used throughout this project, along with their corresponding functions. For further details on this subject, we refer the reader to the instrument's programming manual.

Command or Query	Example	Description
OUTPut	OUTPUT ON	Turn on the output of the instrument
CURRent	CURR 0.1	Change the current value of the current output to *0.1A*
VOLTage:RANGe	VOLT:RANG P25V	Select the *+25V* output channel
CURRent:PROTection	CURR:PROT 0.7	Set limit current value to *0.7A*
*RST	*RST	Reset instrument to default settings
SYSTem:ERRor?	SYST:ERR?	Read any errors from the error buffer
VOLTage	VOLT 12	Change the voltage value of the voltage output to *12V*

Table III.1: Remote PSU commands.

The commands and queries are used within the subVI Agilent_GPIB_FSM.VI. This subVI is launched via an asynchronous call from the second consumer loop within Arduino_FlowPump_Control, which writes high-level commands in human-readable format, and the low-level subVI translates these commands in the proper syntax used by the PSU. The second consumer loop does not know how these commands are transferred to the instrument, once again highlighting the advantage of the information hiding technique.

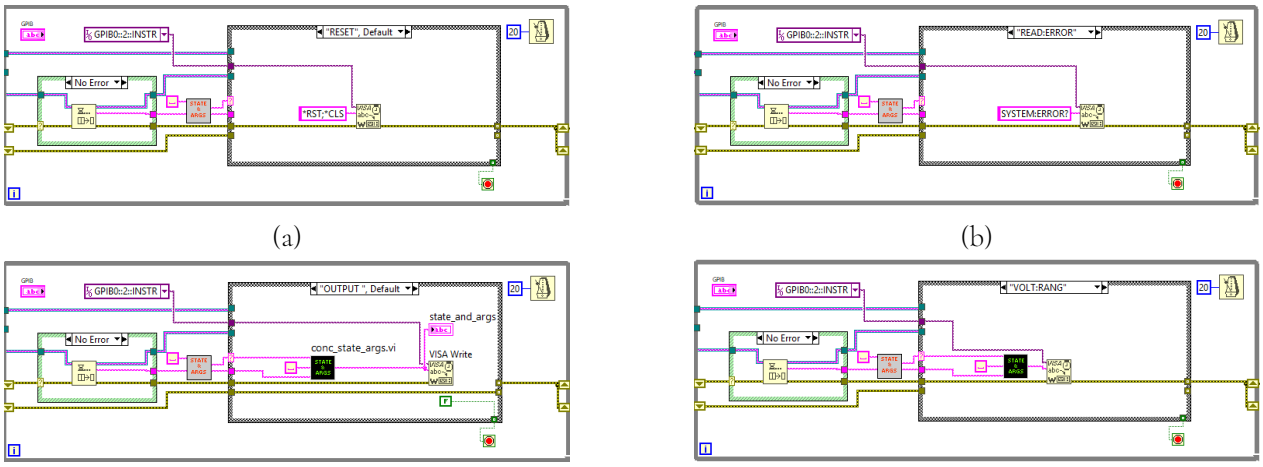


Figure 24: Agilent_GPIB_FSM.vi Block Diagram.

The main functions used from the Instrument I/O / VISA subpalette. VISA subVIs are able to transfer the command desired as prescribed by the interface used, which is identified based on the **VISA Resource Name** used. In this case, since the program uses a **GPIB** resource name, the VISA subVI transfers the commands according to the GPIB protocol, as shown in Figure 24d. Also, note that the low-level subVI uses a **support VI** called *conc_state_arg*. This subVI reconstructs the command string used by the higher-level subVi Arduino_FlowPump_Control.vi.

Current Ramp FSM

The third while loop in the Arduino_FlowPump_Control.vi implements the finite state machine required to model the states of a current ramp that is applied to the air pump, generating the airflow to be measured by the transducers.

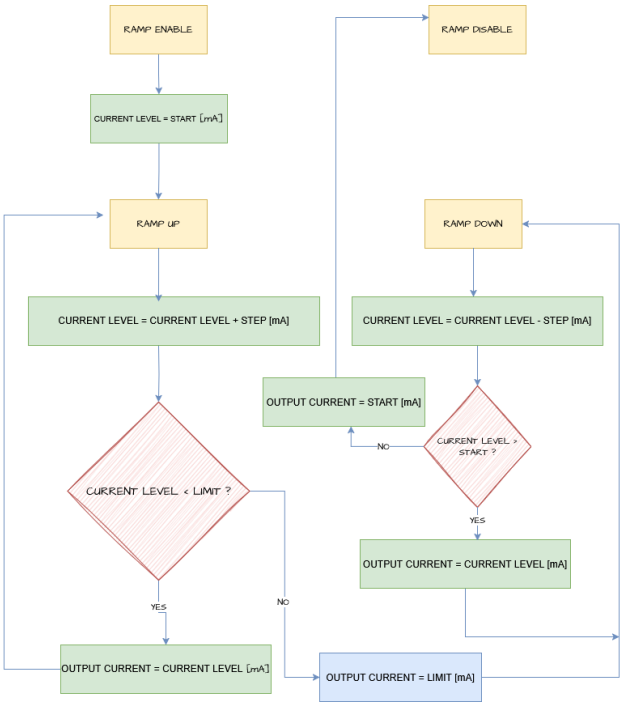
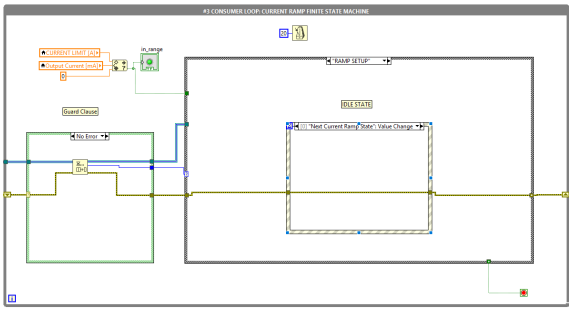


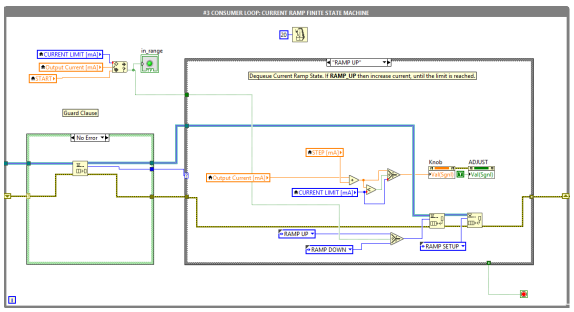
Figure 25: Current Ramp Flow diagram

The diagram shown in Figure 25 effectively illustrates the process of generating a current ramp, but it is not complete in itself, as some states are intentionally omitted for simplicity. However, these omitted states are essential for the correct functioning of the program in practice.

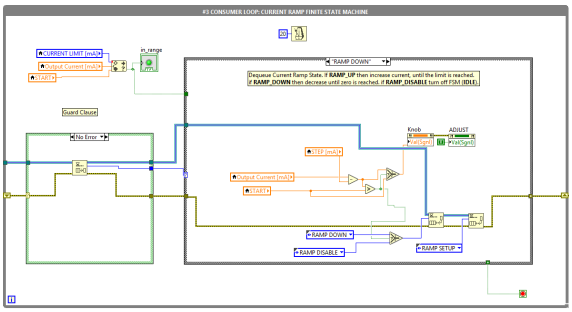
The decision to implement a finite state machine stems from the intent to make the ramping process both manual and iterative. Indeed, if one were to implement a program that strictly followed the flow diagram shown in Figure 25, the execution would proceed automatically and without user intervention, as such a process typically requires no external control.



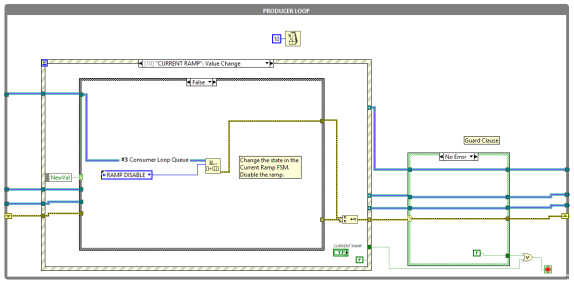
(a) RAMP SETUP state



(b) RAMP UP state



(c) RAMP DOWN state



(d) RAMP DISABLE state

Figure 26: Current Ramp FSM in the Arduino_FlowPump_Control subVI.

In contrast, our design aims to let the user decide when to apply the next current value (*CURRENT VALUE* ± *STEP [mA]*, depending on the state which could either be *RAMP UP* or *RAMP DOWN*), rather than having the algorithm advance automatically. Therefore, a key state—absent from the diagram in Figure 25—must be introduced: a state in which the system waits for a user-issued command before transitioning to the next state, based on a condition evaluated in the previous stage. In the program, this waiting state is referred to as **RAMP SETUP**, shown in Figure 26a.

When the automaton is in the **RAMP UP** or **RAMP DOWN** states —that is, when increasing or decreasing current values are being generated, respectively—two transitions are scheduled. One is conditional, based on a Boolean evaluation; the other transitions to the waiting state. These states are shown in Figure 26b and 26c, respectively.

To implement this mechanism, an event structure similar to the one seen in the Producer loop is inserted into the case structure within the third consumer loop. The difference lies in the fact that we wait for one of two events to occur:

- The user disables the current ramp generation procedure (**RAMP DISABLE**);
- The user requests that the next current value be generated according to the current state, the

current limit, and the selected step.

In the first of these two cases, it is the Producer loop that inserts the current ramp disable state into the state transition queue, as shown in Figure 26d.

Commands Decoder Loop

Finally, we added a fourth while loop in the Arduino_FlowPump_Control.vi, which can be used with a queue initialized through the subVI's terminals, to make the subVI itself programmable by other VIs created by the user. We have indeed defined an **API** with which the user can send strings (*commands*) accompanied by arguments (*parameters*) that generate an event, effectively simulating a real user pressing a push-button or any other possible user-GUI interaction on the front panel. This allows the **main.vi** program to modify the state of the controls on the subVI's front panel without necessarily interacting with the virtual instrument's front panel. Furthermore, this modification allows for an additional level of abstraction; in fact, we have defined commands that involve multiple actions (i.e., multiple interactions between the user and the GUI) that allow more operations to be performed with a single command.

Chapter IV

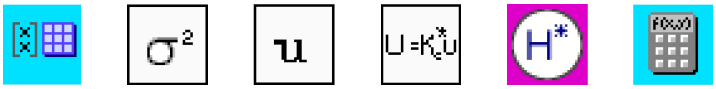
Calibration

Calibration Procedure Loop

To aid the user in the analysis of the datasets acquired and produce a series of results that can be used to produce a calibration certificate, a set of VIs was developed for this aim. These subVIs are used within the **main.vi** block diagram, and are executed once the user loads the datasets in the application after the data acquisition phase. In the main.vi a fourth while loop executes whenever the user loads the acquired datasets during the current ramp and data acquisition procedures. In total, the user must acquire two datasets for each phase (ascending and descending input sequence) for both sensors.

A set of two dedicated **LabVIEW libraries** was created to group the subVIs related to a specific task in the calibration procedure task: **statistics.lvlib** and **CALIBRATION.lvlib**. The subVI Icons are shown in Figure 27.

(a)

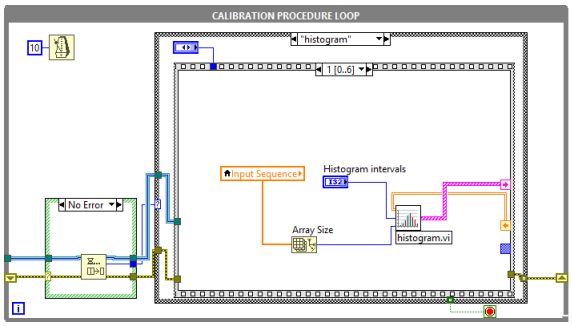


(b)

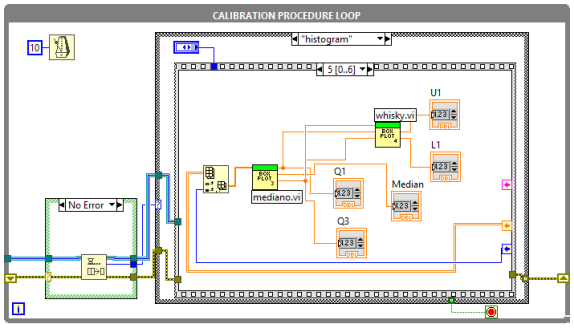
Figure 27: (a) statistics.lvlib subVIs. (b) CALIBRATION.lvlib subVIs.

Samples Statistical Distribution

Before evaluating Type A Uncertainty, it is essential to check the probability distribution of the sample data in each dataset, as this helps determine whether assumptions underlying statistical models—such as normality—are valid.



(a) Evaluation of the histograms from the datasets loaded by the user after acquisition

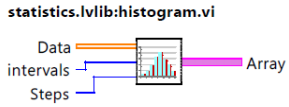


(b) Search for median, lower and upper quartiles (Q1, Q3) and whiskers (U1, L1).

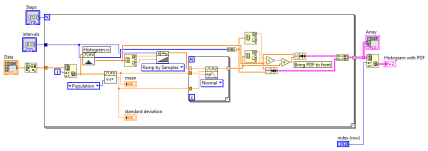
Figure 28: Calibration Procedure Loop: histogram and boxplots graphs and statistical metrics computation.

Figure 28 provides an overview of the statistical distribution analysis step:

- **(a)** shows how the **histograms** get created, thanks to the **histograms.vi** in the statistics.lvlib, whose block diagram is shown in Figure 29b. Note that a histogram is created for each column in each dataset (ascending or descending) for both sensors. Each column represents a **feature**, which is a value of output current that drove the flow pump during the acquisition phase.
- **(b)** shows how the median value, lower and upper quartiles (Q1, Q3) and whiskers (U1, L1) are computed thanks to the **mediano.vi** and **whisky.vi** subVIs in the statistics.lvlib, whose block diagram are shown in Figure 30b and 31b. The first subVI computes the median to divide the ordered dataset (a feature column) into two halves. The media is not included in either half. The lower quartile value is the median of the lower half of the data. The upper quartile value is the median of the upper half of the data.

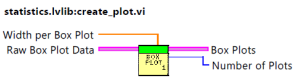


(a) histogram.vi SubVI Description

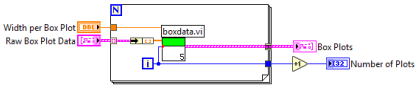


(b) histogram.vi SubVI Block Diagram

Figure 29: **histogram.vi** subVI used to create all the histograms for each feature column in a given dataset

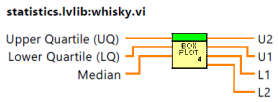


(a) create_plot.vi SubVI Description

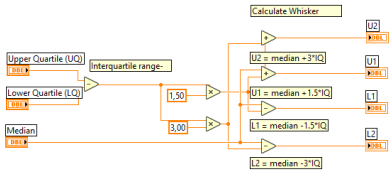


(b) create_plot.vi SubVI Block Diagram

Figure 30: **create_plot.vi** subVI used to create all the boxplots for each feature column in a given dataset

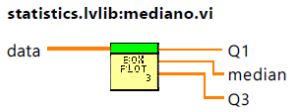


(a) whisky.vi SubVI Description

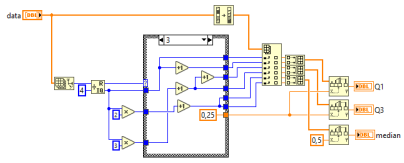


(b) whisky.vi SubVI Block Diagram

Figure 31: **whisky.vi** subVI used to compute the lower and upper whiskers for each feature column in a given dataset



(a) mediano.vi SubVI Description

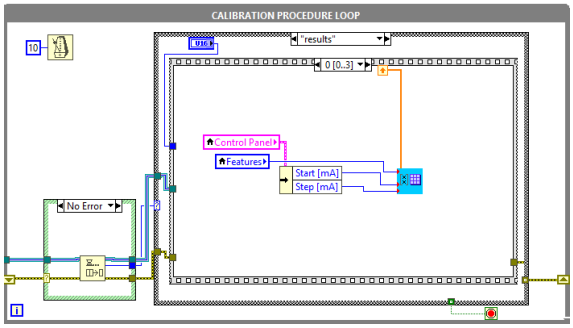


(b) mediano.vi SubVI Block Diagram

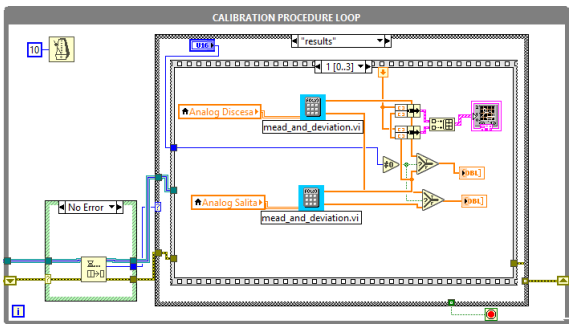
Figure 32: **mediano.vi** subVI used to compute the lower and upper quartiles and the median value for each feature column in a given dataset

Expected Value and Standard Deviation

The next step is the evaluation of the expected values from the sensors datasets acquired during both ascending and descending phases. This is achieved in the **results** state in the *Calibration Procedure Loop*.



(a) Search for the number of input values (current values) used during the acquisition phase

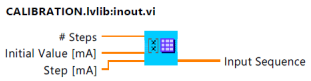


(b) Search samples mean and standard deviations from samples acquired for analog sensor datasets.

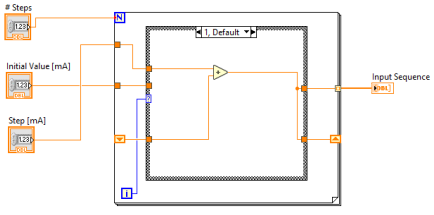
Figure 33: Calibration Procedure Loop: expected values and standard deviations evaluation phase.

Figure 33 provides an overview of the first results state:

- **(a)** This frame demonstrates how the loop processes the input sequence (current values driving the air-pump), which is generated by the **inout.vi** subVI (from *CALIBRATION.lvlib*, detailed in Figure 34b). Derived from user-defined parameters like step and starting current values (Figure 18), this sequence is subsequently used to generate graphs of the ascending and descending curves for both sensors.
- **(b)** illustrates the updating process of the indicators shown on the front panel that plot ascending and descending curve values for the Analog Sensor. The values computed are *expected values* values and *standard deviations* **point-by-point**. These values are computed by the subVI **mean_and_deviation.vi** in the CALIBRATION.lvlib, whose block diagram is shown in Figure 35b.

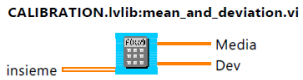


(a) inout.vi SubVI Description

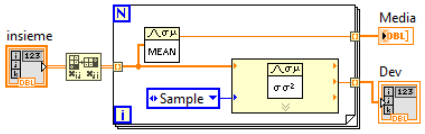


(b) inout.vi SubVI Block Diagram

Figure 34: **inout.vi** subVI used to compute the input sequence values used during the acquisition phase, which is, the number of feature columns in each dataset



(a) mean_and_deviation.vi SubVI Description



(b) mean_and_deviation.vi SubVI Block Diagram

Figure 35: **mean_and_deviation.vi** subVI used to compute *expected values* values and *standard deviations* in feature columns in each dataset

Standard and Expanded Uncertainty

The next step verifies the presence of Hysteresis in the sensors measurements. To do this the maximum standard uncertainty from both the ascending and descending dataset has to be evaluated. For this reason, in this section, the method for the evaluation of the standard and combined uncertainty is presented.

Once a sufficient measurements of air flux and voltage are taken, both in ascent and descent phase, the following procedure is applied for both the analog and digital sensor:

1. The mean $\bar{\Phi}$ of the measurements:

$$\bar{\Phi} = \frac{1}{N}\sum_{i=1}^{N}\Phi_i$$

   where N is the number of measurements and Φ_i are the individual air flux measurements.

2. The standard deviation σ_Φ of the measurements:

$$\sigma_\Phi = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(\Phi_i - \bar{\Phi})^2}$$

3. The **standard uncertainty** u_Φ, calculated as the standard deviation divided by the square root of the number of measurements N:

$$u_\Phi = \frac{\sigma_\Phi}{\sqrt{N}}$$

4. After calculating the standard uncertainty for each air flux and voltage measures, the maximum value of standard uncertainty in the upward direction U_{up} and the downward direction U_{down} is determined.

5. The maximum expanded uncertainty , obtained by multiplying the maximum uncertainty by the coverage factor k = 2:

$$U_\Phi = k * u_{max}$$

6. The maximum deviation between the uphill and downhill phase, Δ_{max}, is:

$$\Delta_{max} = max\{|\Delta_i|\}$$

where Δ_i is given by:

$$\Delta = |\bar{\Phi}_{up_i} - \bar{\Phi}_{down_i}|$$

In words, Δ_i is the i-th absolute deviation between the ascending curve i-th expected value ($\bar{\Phi}_{up_i}$) and the descending curve i-th expected value ($\bar{\Phi}_{down_i}$), computed for each operating point tested generating a different current amplitude.

7. Finally, the combined uncertainty $U_{combined}$ was calculated as the square root of the sum of the squares of the two maximum expanded uncertainties:

$$U_{combined} = \sqrt{U_{up}^2 + U_{down}^2}$$

8. The same procedure is applied for the analog sensor voltage measurements.

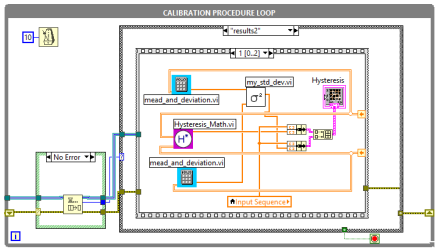
Hysteresis Validation

The program then assesses sensor measurements for **Hysteresis**, a critical nonlinearity indicating a memory effect where output depends on input history. Detecting hysteresis is vital as it causes discrepancies between measurements at identical input values during ascending versus descending ramps, potentially biasing uncertainty evaluations. Numerical assessment involves comparing mean sensor outputs at corresponding input values across ascending and descending phases. If this difference exceeds a predefined threshold relative to sensor resolution or noise, hysteresis is confirmed. Figure 36 provides an overview of the second results state:

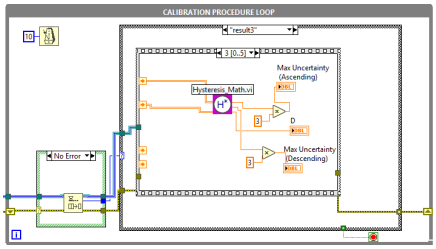
- **(a)** shows how the **Hysteresis_math.vi** included in the CALIBRATION.lvlib, and whose block

diagram is shown in Figure 37b. This subVI computes the **point-by-point deviation** between the ascending and descending curves, while the **my_std_dev.vi** computes the **the standard deviation**. This subVI's block diagram is shown in Figure 39b.

- **(b)** illustrates how the subVI **Hysteresis_math** is used to compute the **maximum uncertainty** during the ascending and descending phase, as well as the **maximum difference** between the curves.

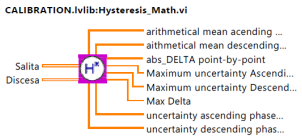


(a) Compute deviation between the ascending and descending curves plus the combined point-by-point deviation resulting from ascending and descending curves.

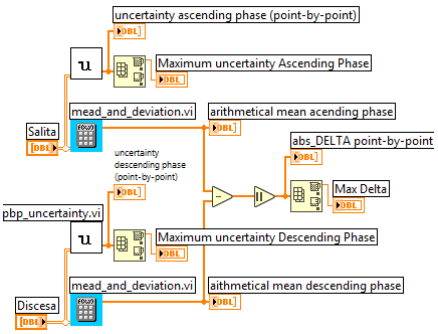


(b) Evaluation of the maximum deviation between the ascending and descending curves of the sensor's datasets and their maximum expanded uncertainty during each phase.

Figure 36: Calibration Procedure Loop: Hysteresis evaluation phase.

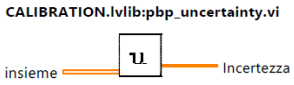


(a) Hysteresis_math.vi SubVI Description

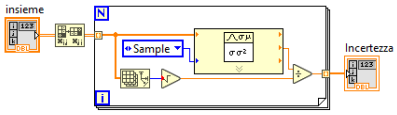


(b) Hysteresis_math.vi SubVI Block Diagram

Figure 37: **Hysteresis_math.vi** subVI used to compute the relevant mathematical information to discriminate whether the sensors' measurements are affected by hysteresis.

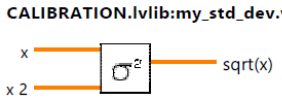


(a) pbp_uncertainty.vi SubVI Description

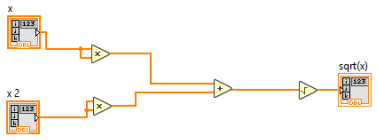


(b) pbp_uncertainty.vi SubVI Block Diagram

Figure 38: **pbp_uncertainty.vi** subVI used in the Hysteresis_math.vi to compute the **point-by-point standard uncertainty**.



(a) my_std_dev.vi SubVI Description

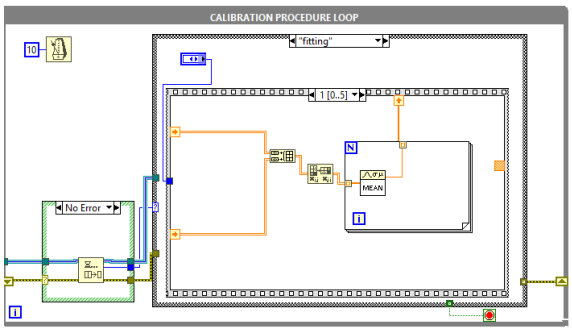


(b) my_std_dev.vi SubVI Block Diagram

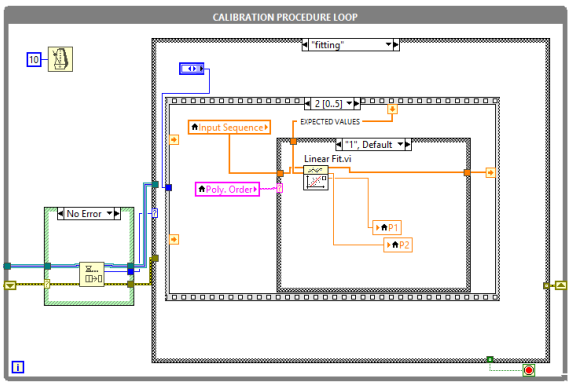
Figure 39: **my_std_dev.vi** subVI used to compute the combined standard deviation, point-by-point, between the ascending and the descending curves.

Curve Fitting

After having evaluated the standard uncertainty for each sensor measurement, and having verified if this metric should also account for the phenomenon of Hysteresis, the next step is to find the best-fit estimators to the data curves derived from the sensors' dataset. This will enable to prediction of the sensors' output values starting from the current value directed to the air-pump that has been used to generate the air-flow.



(a) Compute the expected values from the combined ascending and descending datasets

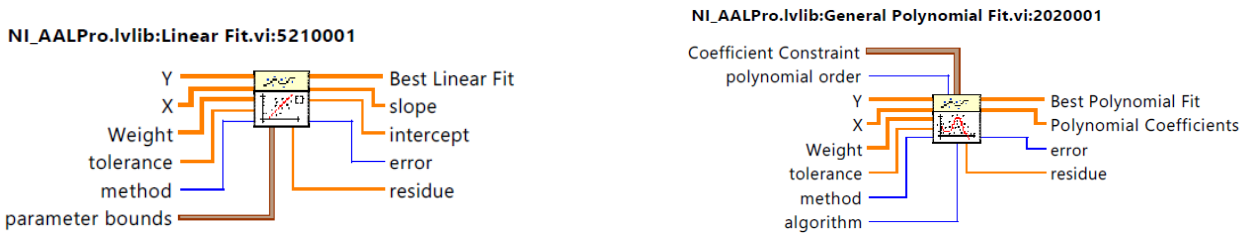


(b) Search for Best Estimator, depending on polynomial order choosed.

Figure 40: Calibration Procedure Loop: curve fitting phase.

Figure 40 provides an overview of the fitting state:

- **(a)** represents how the combined dataset- resulting from the concatenation of the ascending and descending datasets- are used to compute the expected values for the given sensor choosed for the curve-fitting procedure.
- **(b)** illustrate how the best estimator is searched for, depending on the polynomial order chosen by the user on the front panel. The user can either select a first or second order polynomial. Accordingly, different subVIs from the **Mathematics/Fitting Palette** will be used. In particular, in the first the **Linear Fit.vi** will be used for the first order polynomial, and the **General Polynomial Fit.vi**, with polynomial order equal to two, in the second case.



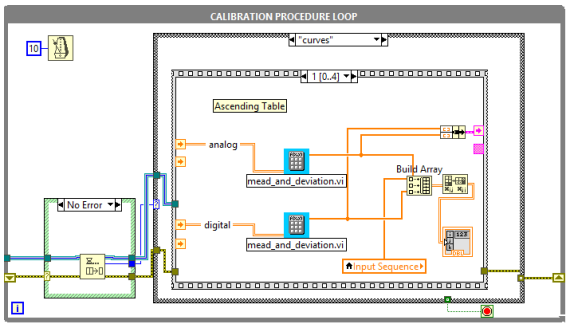
(a) Linear Fit SubVI Description

(b) General Polynomial Fit SubVI Description

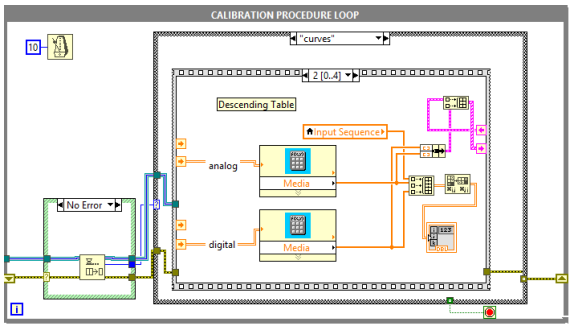
Figure 41: Curve Fitting subVIs used to compute the **best-fit** curves that can predict the sensors' outputs given the input values (current values used to power the air-pump).

Calibration Curve

Finally, the program computes the conversion equation that will allow to conversion between the voltage output of the analog sensor into a flow measurement comparable with that of the calibrated reference sensor measurements. First, though, the program produces a comparison between the sensors' measurements. This comparison is achieved in the form of tabulated values and graphs that compare the sensors' response in correspondence with the same input value produced at the air-pump terminals.



(a) Ascending Curves Comparison Table



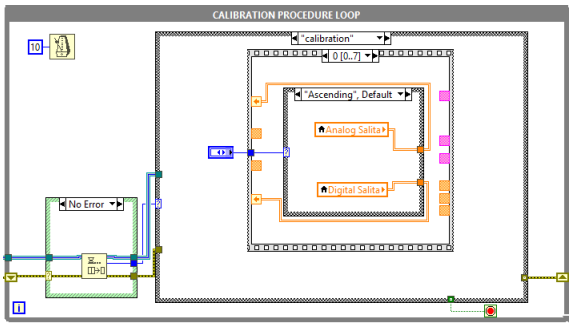
(b) Descending Curves Comparison Table

Figure 42: Calibration Procedure Loop: comparison phase.

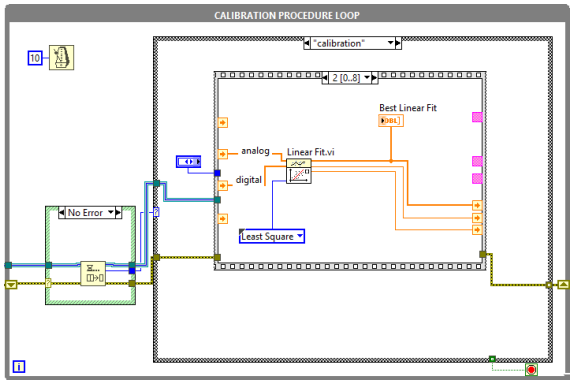
Figure 42 provides an overview of the comparison state:

- **(a)** shows the frame where the datasets resulting from the ascending phase are used to compute the expected values in this case for both sensors, respectively. A table is produced, where the first column lists the input current values used during the acquisition, the second column lists the analog sensor expected values, and the third column lists the digital sensor expected values.
- **(b)** the same is accomplished for the descending phase. Also, note that the columns in the two tables are used to create clusters that will be used to plot on a XY Graph the comparison between the sensors

The last step is the computation of the calibration curve and the validation of the estimator's goodness.



(a) Analog and Digital Table selection.



(b) Best LS Linear Estimator Search.

Figure 43: Calibration Procedure Loop: calibration phase.

Figure 43 provides an overview of the calibration state:

- **(a)** shows the frame where the analog and digital datasets resulting from the ascending phase or descending phase are seelected. The goal is to find a linear estimator that from the raw voltage output of the analog sensor can convert to a mass flow rate measurement, comparable with that of the reference digital flow sensor.
- **(b)** The **X** terminal of the Linear Fit.vi is connected to the digital reference sensor expected values, while the **Y** terminal is connected to the expected values of the analog sensor under test.

Calibration Report

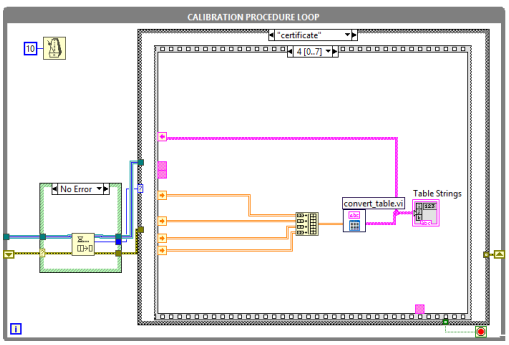
A Certificate of Calibration is a formal document that attests to the accuracy and performance of a measuring instrument or device. It serves as official proof that the instrument has been calibrated against traceable standards, ensuring its measurements are reliable and meet specified requirements.

In the context of our project, the certificate of calibration would document the performance of your Renesas FS-1012-1100NG analog thermal mass flow sensor (the Device Under Test - DUT). It would confirm that the sensor's voltage output corresponds accurately to specific flow rates in Liters/minute. An extensive uncertainty budget was developed, combining Type A uncertainty (from repeatability) with several Type B sources, including the digital reference sensor's accuracy (% of reading), the DAQ's resolution, the power supply's resolution, and the uncertainties of the calibration curve's angular coefficients and offsets. All uncertainties were combined using Root Sum of Squares, with sensitivity coefficients used to handle different units.

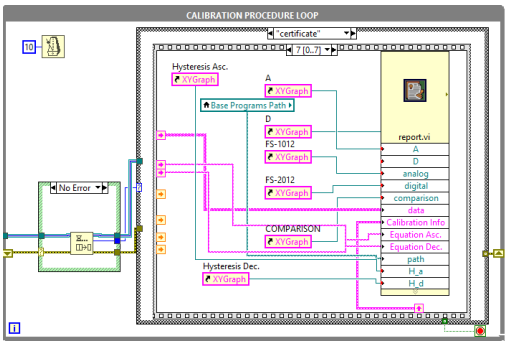
A dedicated LabVIEW Library and two subVIs were created for the purpose of reporting the results of the calibration procedure: ***convert_table.vi*** and ***report.vi***. These subVI's icons are shown below:

Figure 44: **Certificate.lvlib** subVIs. From left to right: *convert_table.vi*, *report.vi*

The *convert_table.vi* is used within the Calibration Procedure Loop to convert the table composed of numerical values that summarises the results obtained from the calibration dataset into a string-valued table. The *report.vi* uses the functions from the **Report Generation** subpalette to compile a **template calibration report** that is included in the project's files folder.



(a) Selection of the datasets, and conversion to string valued table.

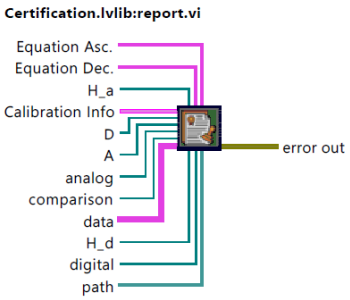


(b) Connection of the required controls to the subVI **report.vi**.

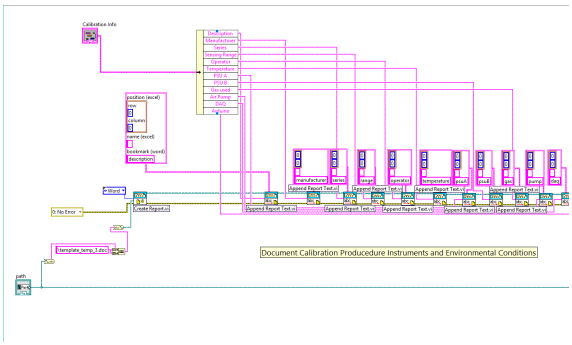
Figure 45: Calibration Procedure Loop: third step into the results evaluation phase.

Figure 45 provides an overview of the certificate state:

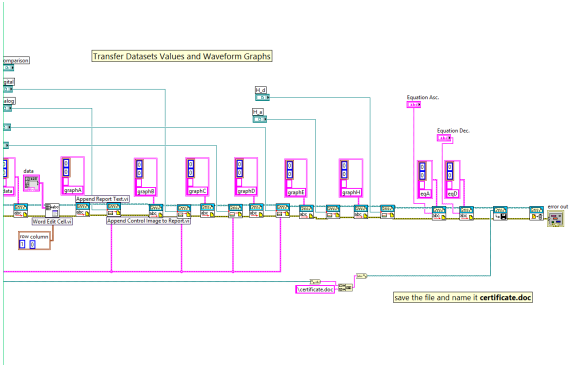
- **(a)** shows the frame where the datasets resulting from the ascending phase, descending phase from both sensors are combined together and converted into a string-valued table format.
- **(b)** the subVI report.vi is used to generate the calibration report using various controls that are found on the **main.vi** front panel. For each graph that has to be printed in the calibration report, a **reference** to the relative indicator was created. This reference is used in the report.vi block diagram, connecting it to the terminal **ctrl reference** of the **Append Control Image To Report.vi**. The subVI diagram is shown in Figure 45.



(a) **report.vi** subVI Description

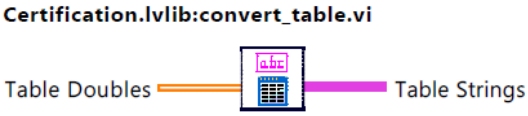


(b) **report.vi** block diagram. Calibration Informations gets transferred into the Report Template.

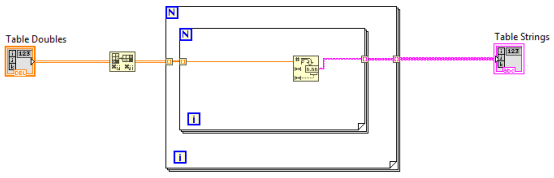


(c) **report.vi** block diagram. The front panel graphs are printed in the calibration report.

Figure 46: **report.vi** subVI used in the calibration procedure loop to generate the calibration report to document the results obtained from the calibration datasets.



(a) convert_table.vi SubVI Description



(b) convert_table.vi SubVI Block Diagram

Figure 47: **convert_table.vi** subVI used to convert a double-valued table into a string-valued table.

Chapter V

Results

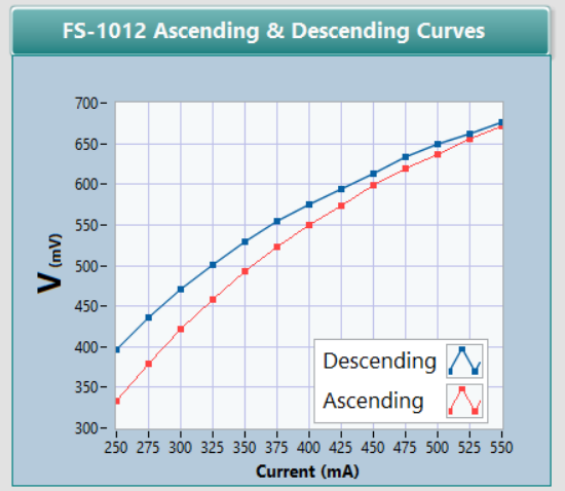
This chapter outlines the results found during the data analysis procedure of the calibration dataset collected. This analysis has been **entirely** conducted in the **LabVIEW** development environment, thanks to the analysis tools offered by the main.vi *Calibration Procedure Loop*, presented in the ***Calibration*** chapter of this manuscript.

Calibration Dataset Analysis

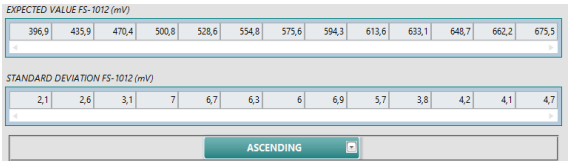
The ascending and descending curves generated during the analog sensor's calibration phase reveal a significant difference in output voltage depending on whether the input is varied in an increasing or decreasing manner. This same behavior has been observed and can be generalized to the digital sensor's calibration datasets as well.

When combining the curves from both sensors (in both ascending and descending phases), a direct relationship emerges between the outputs of the two instruments.

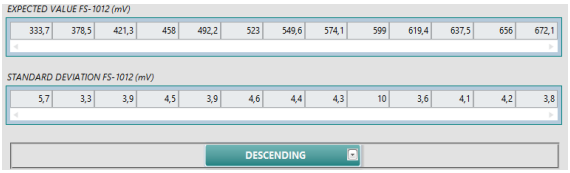
Curves, Expected Values and Standard Deviation



(a)

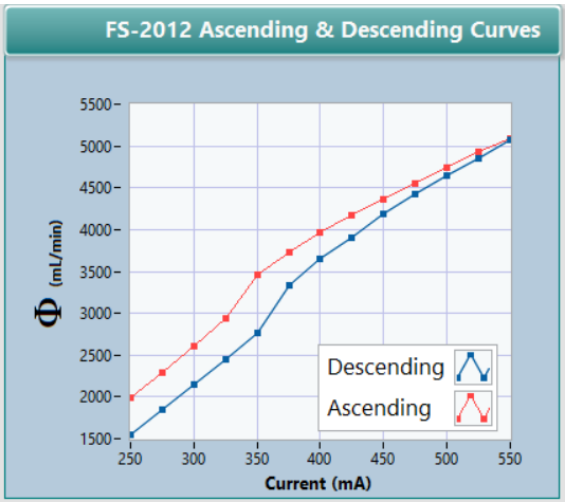


(b)

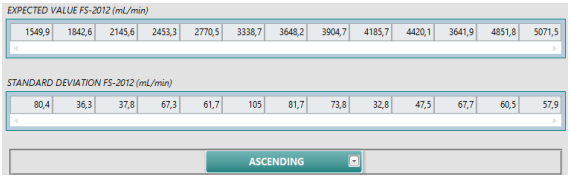


(c)

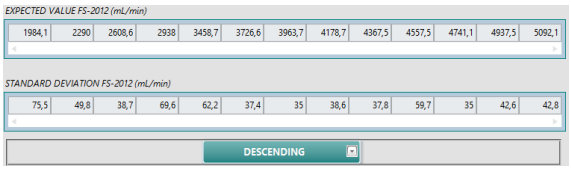
Figure 48: FS-1012 ascending and descending curves(a) with respective expected values (b) and standard deviations (c)



(a)



(b)



(c)

Figure 49: FS-2012 ascending and descending curves(a) with respective expected values (b) and standard deviations (c)

Figures 50 and 51 provide a detailed performance analysis of two sensors, the FS-1012 and FS-2012, through their calibration curves and associated statistics.

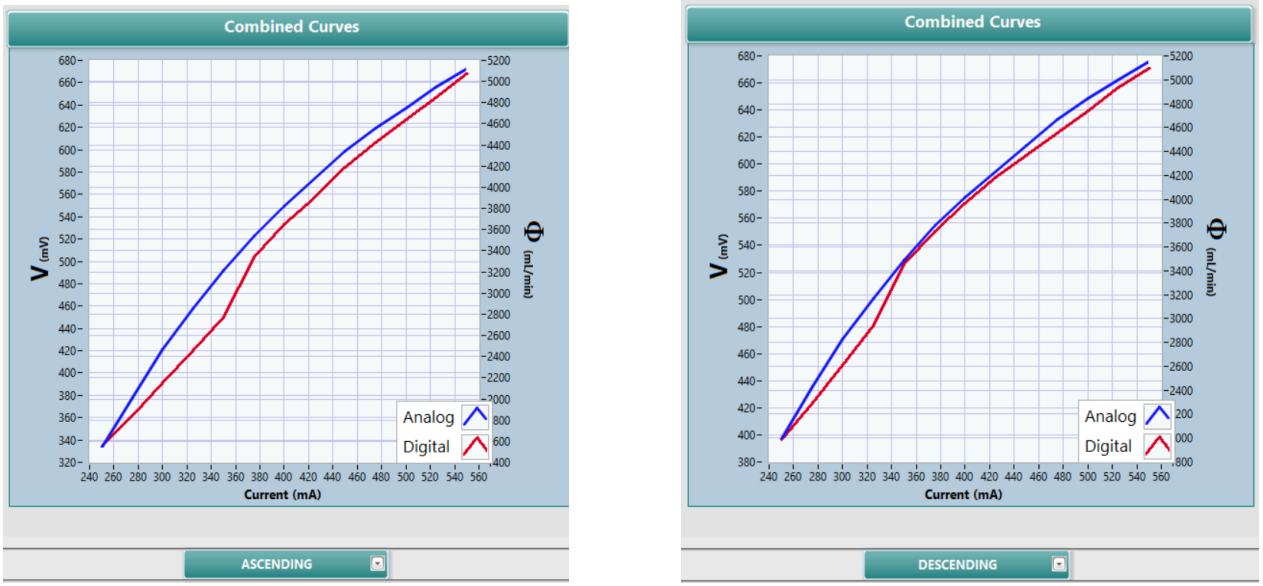
- Figure 50 focuses on the FS-1012 sensor:

- Graph (a) illustrates the up- and down-curves that correlate voltage (V in mV) with current (mA), highlighting the sensor's hysteresis.
- Tables (b) and (c) complement the graph by providing, respectively, the expected voltage values and standard deviations corresponding to the various measured currents, for both the up- and down-curves.

• Similarly, Figure 51 (which follows the same structure) examines the FS-2012 sensor:

- Graph (a) displays the up- and down-curves, showing the relationship between flow (Φ in mL/min) and current (mA).
- Tables (b) and (c) provide the numerical values. expected and standard deviations of the flow for the different currents, completing the quantitative analysis of the sensor behavior.

Comparison Graph



(a) (b)

Figure 50: Combined curves for ascending (a) and descending (b)

The (a) Ascending and (b) Descending graphs in Figure 52 combine both the analog and digital signals as a function of current (mA) on the same scale. This allows the response of both outputs to current

variations to be displayed simultaneously, providing a complete picture of the overall system behavior during the current increase and decrease phases.

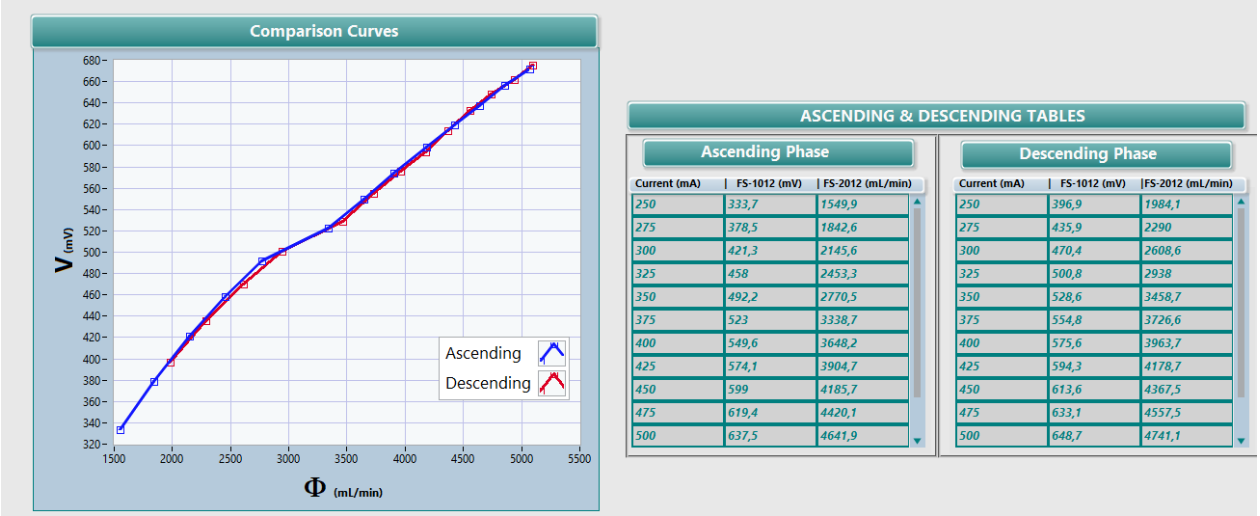
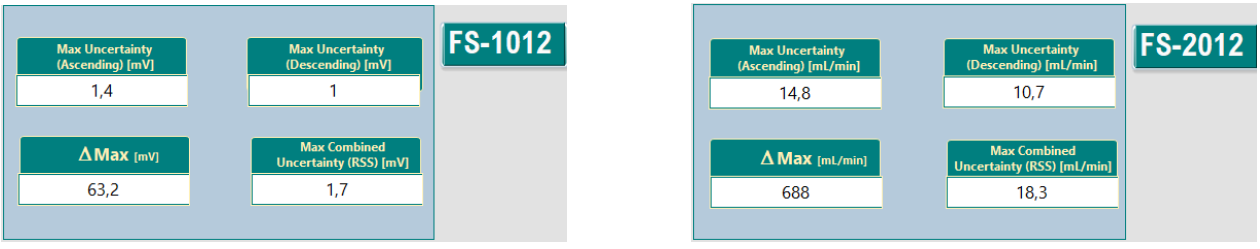


Figure 51: Comparison curves with ascending and descending curves

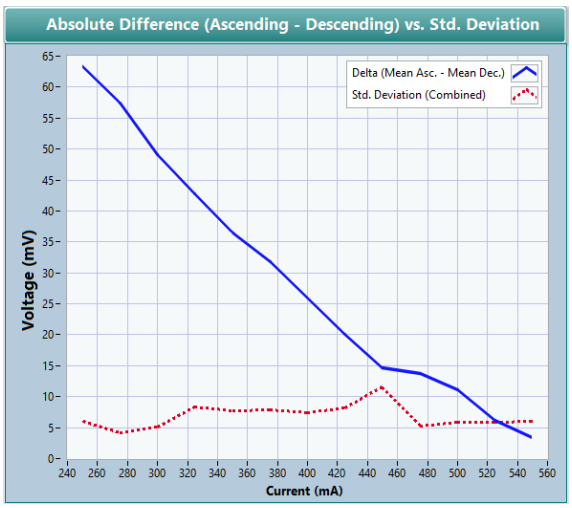
Figure 53 presents a comparative analysis of sensor performance. The left graph shows ascending and descending curves, illustrating the relationship between Voltage (V) and Flux (Φ). Next to the graph, the "ASCENDING and DESCENDING TABLES" provide the measured values (FS 1012 (mV) and FS 2012 (mL/min)) that were detected at specific current (mA) values, for both the ascending and descending phases. This combination of graph and tables provides a comprehensive and quantitative view of the sensor's behavior under different operating conditions.

Hysteresis Verification

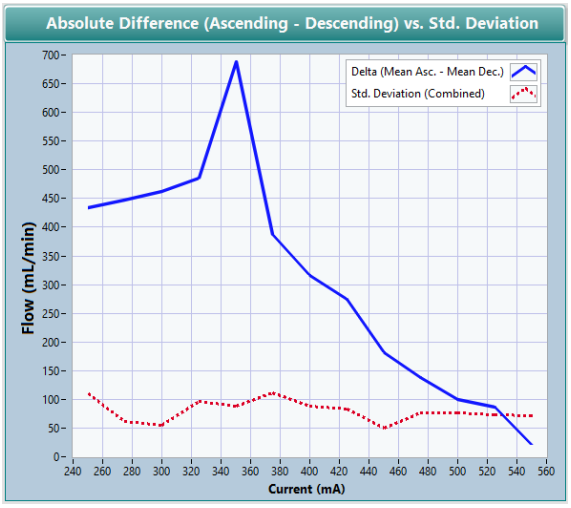


(a) (b)

Figure 52: Hysteresis: Numerical Comparison, Maximum Deviation and Maximum Combined Uncertainty (RSS) FS-1012 (a) and FS-2012 (b)



(a)



(b)

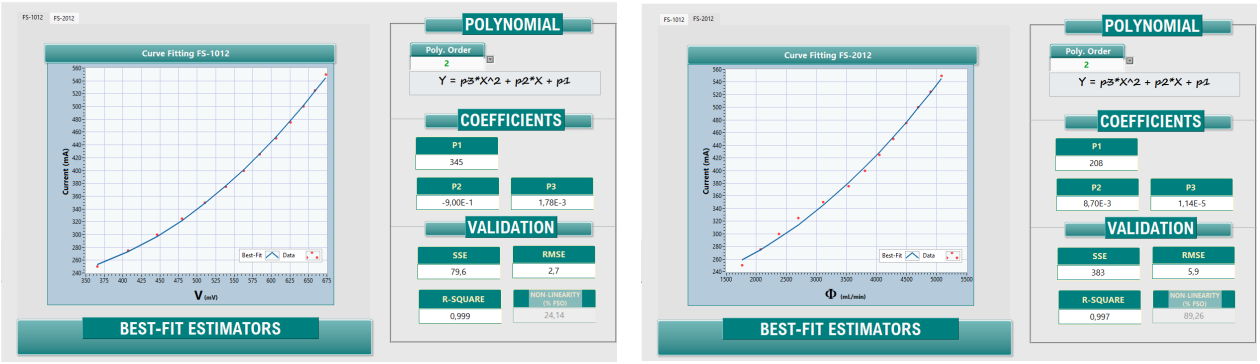
Figure 53: Hysteresis: Comparison Graph, Maximum Deviation between two points and the Combined Standard Deviation (RSS) FS-1012 (a) and FS-2012 (b)

From the values obtained it was possible to carry out a detailed analysis of the uncertainties and behaviour characteristics of the two sensors, in particular:

- Fs-1012: The **maximum standard uncertainty** for ascending is **1.4 mV**, while for descending it is **1 mV**. Both these values are ***expanded uncertainties*** computed with a 95% confidence level, using a coverage factor equal to 2. The **maximum combined uncertainty** (obtained using the RSS method) is **1.74 mV**. The **maximum deviation recorded between two points is 63.2 mV**, indicating the presence of **hysteresis** in the system.
- FS-2012: The **maximum standard uncertainty** for ascending is **14.8 mL/min**, while for descending it is **10.7 mL/min**, both computed with a 95% confidence level, using a coverage factor equal to 2. The **maximum combined uncertainty** (obtained using the RSS method) is **18.3 mL/min**. The **maximum deviation between two points is 688 mL/min**, indicating **hysteretic** behavior for this sensor as well.

Best-Fit Estimators

Figures 56 and 57 present the polynomial fitting analysis for two sensors, FS-1012 and FS-2012, providing a mathematical model for their calibration curves and the related validation metrics. Specifically,

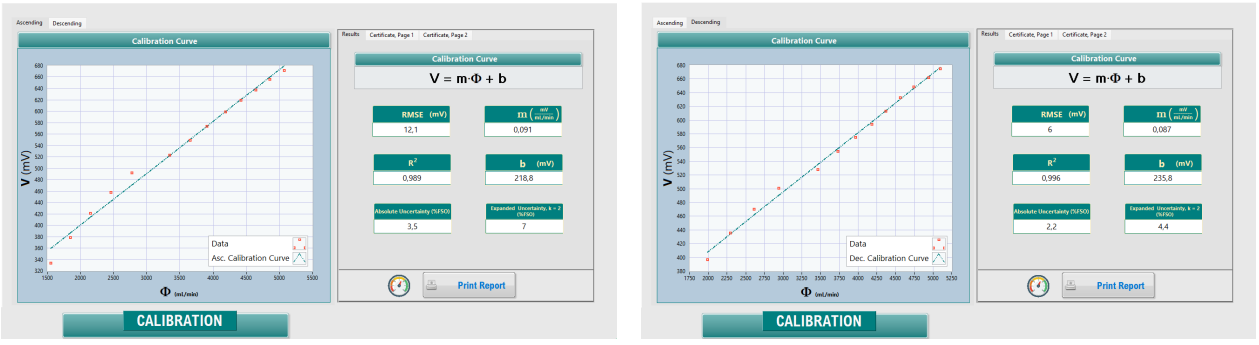


(a) (b)

Figure 54: Best-Fit Estimators: FS-1012 (a), FS-2012 (b)

Fig. 56 focuses on the FS-1012 sensor, graphically showing the correlation between Current (mA) and Voltage (V in mV) with the polynomial fitting curve. The adjacent section reports the details of the (second-degree) polynomial, including the coefficients (p1, p2, p3) and performance indicators such as SSE, RMSE, R-Square and Nonlinearity. Similarly, Figure 57 analyzes the FS-2012 sensor, illustrating the fitting of the curve relating Current (mA) to Flow (Φ in mL/min). For this sensor, the order of the polynomial, its coefficients and the validation metrics are also specified, allowing a complete evaluation of the accuracy of the model."

Calibration Curves



(a) (b)

Figure 55: Calibration Curve: Linear Regression Estimator Graph, Ascending Dataset (a) and Descending Dataset (b)

The Calibration Curve is represented by the equation:

$$V = m \cdot \Phi + b \tag{V.1}$$

For both the Ascending Phase and Descending Phase Datasets, a Linear Regression Estimator was used. In the case of the Ascending Phase dataset, the calibration curve results in:

$$V = 0.091 \cdot \Phi + 218.8 \tag{V.2}$$

The coefficient of determination, R^2, is calculated to be 0.989. This value, although high, is evidence of slight curvature, hence deviation of collected data from the linear model. Nevertheless, the goodness of the estimator was considered sufficient for the purpose. In the case of the for the Descending Phase Dataset, the calibration curve is the following:

$$V = 0.087 \cdot \Phi + 235.8 \tag{V.3}$$

The coefficient of determination, R^2, is calculated to be 0.996. This high value indicates an excellent fit of the calibration curve to the measured data, suggesting that the model reliably represents the relationship between the inputs and outputs.

Automated Measuring System Uncertainty Budget

As presented in the ***Calibration*** chapter of this manuscript, the ***main.vi*** program is capable of producing a ***calibration report***. In this context, the final result from the calibration procedure is produced: the ***Combined Standard Deviation*** of the Measuring System. This value arises from the ***Uncertainty Budget***, which considers all the relevant ***Sources of Uncertainty*** in the Automated Measuring System. From the known distribution of each source, a ***standard uncertainty*** is evaluated.

Source Of Uncertainty	Estimate	Distribution	Unc. Contribution	Unit
DAQ Resolution	4.88	Type B, Rectangular	2.8	mV
DUT Standard Error of the Mean	-	Type A, Normal	0.7	mV
Regression Fit Intercept Error	218.8	Type A, Normal	23.4	mV

Table V.1: Uncertainty Budget, Ascending Phase Dataset.

Measuring System Combined Standard Uncertainty: 3.5%

Expanded Uncertainty $(k = 2)$**: 7%**

Source Of Uncertainty	Estimate	Distribution	Unc. Contribution	Unit
DAQ Resolution	4.88	Type B, Rectangular	2.8	mV
DUT Standard Error of the Mean	-	Type A, Normal	0.5	mV
Regression Fit Intercept Error	235.8	Type A, Normal	14.5	mV

Table V.2: Uncertainty Budget, Descending Phase Dataset.

Measuring System Combined Standard Uncertainty: 2.2%

Expanded Uncertainty $(k = 2)$**: 4.4%**

Among the sources of uncertainty that were identified in the measuring systems we account for the ***DAQ Resolution***, the **Calibration Fitting Curve Intercept Standard Error**, and the **Standard Error of the Mean** derived from the transducers datasets. These values are combined with the ***Root Sum of Squares*** method. Finally, the obtained value is expressed as a percentage of the ***Full Scale Output (FSO)***.

Two uncertainty budgets are produced, one for each dataset (Ascending Phase and Descending Phase). In the case of the Ascending Phase Dataset, the **Combined Standard Uncertainty** is quantified to $u_c = \mathbf{3.5\%}$. The ***Expanded Uncertainty***, consequently, quantifies to $u_\epsilon = \mathbf{7\%}$. In the case of the Descending Phase Dataset, the **Combined Standard Uncertainty** is quantified to $u_c = \mathbf{2.2\%}$. The ***Expanded Uncertainty***, using a coverage factor equal to 2, quantifies to $u_\epsilon = \mathbf{4.4\ \%}$.

Chapter VI

Conclusions

Project Workflow

This project's aim was the development of an automated measurement system for characterizing an air flow transducer, specifically the **Renesas FS-1012-1100-NG** thermal mass analog flow sensor. The calibration has been carried out versus a calibrated digital air flow sensor, precisely the **Renesas FS-2012-1100-NG** thermal mass digital flow sensor. The *project workflow* involved the following steps:

- the signal acquisition from the FS1012 transducer signal conditioning circuit;
- the signal acquisition from the FS2012 transducer signal conditioning circuit;
- Calibration of the FS1012 transducers versus the calibrated FS2012 transducer;
- Control of the PP2 air pump by means a programmable benchtop laboratory power supply;
- Display of the air flow from the FS1012 transducer and the FS2012 transducer in real-time.

Nevertheless, these tasks were addressed only after having considered the physical quantities that have to be dealt with in the measuring system, such as the output voltage of the **Device Under Test (DUT)**. This quantity needed careful consideration before it could be used to represent the variation of the air flow rate generated by the air pump. Indeed, when measured by a multimeter, the DUT's **Full Scale Output (FSO)**– at the maximum air flow rate allowed by the potential risk of overheating the air pump– quantifies to a few millivolts at most. This value depends on the difference between the sensor's thermopiles voltages (TP1 & TP2). Thus, a **conditioning circuit** was designed and built on a soldering

board with the aim of amplifying the two voltage sources separately and then computing their difference. This operation provided a feasible voltage magnitude to be acquired by a data acquisition board.

To carry out the tasks required to calibrate the DUT, a set of software applications was developed within the **LabVIEW** development environment, with the ***main.vi*** being the primary *Virtual Instrument* to be used for the calibration procedure.
The set of device connection statuses (establish, write command, read data, etc.) that had to be modelled for each measuring instrument interface led to the choice of the ***Finite State Machine (FSM)*** software paradigm, together with the ***Producer-Consumer*** design pattern. This choice, revealed later to be a significant advantage, since it enabled to manage each device connection separately and in parallel with the others. Moreover, it significantly decreased the application response time for every user request via the *graphical user interface (GUI)* developed. The latter, also known as the VI ***front panel***, has been designed to guide the user through the calibration procedure, exposing only the relevant controls that should be used for each step in the process, hiding other parameters in dedicated subfolders. The description of the GUI developed for the main.vi is given in the User's Manual that has been written for this project's software application.

For each user interaction (***event***), with the GUI an application behaviour was defined to manage the user request, represented by the event source ("Read button pushed", "Connect to Arduino", etc.). The ***Queue*** software structure was used in order to define an ordered sequence of operations to handle each user request.

The **main.vi** is thought the be used by the user to control the air pump- via the Power Supply Unit (PSU) controlled remotely- and the two air flow rate transducers: the DUT and the reference digital sensor. Three subVIs, each implementing a FSM, were used to handle the connection with the remote PSU (**GPIB connection**), the connection with DAQ board (**DAQmx Tasks**), and the Arduino board (**Serial connection**), respectively. Both the ***Agilent_GPIB_FSM.vi*** and ***Arduino_Serial_FSM.vi*** use the ***VISA (Virtual Instrument Software Architecture)*** subpalette. This subpalette provides a unified, device-independent application programming interface for communicating with various types of instruments and devices, and it is used to send commands and retrieve data from GPIB and Serial device interfaces. The ***daq_FSM.vi***, instead, uses the set of functions from the ***NI DAQmx*** subpalette. This subpalette, allows to control the DAQ various types of measurements or signal generations, including the

analog input reading operation, needed to capture the DUT's output voltage.

When executed, the main.vi uses the **Start an Asynchronous Call** function to call each of these programs during the initialization phase of the application. This means that the application does not wait for the subroutine to end, but instead continues to execute in parallel to the processes it calls. This is a further improvement in terms of program response time, and it has been possible thanks to the Queue Operations used.

Finally, a set of subVIs was developed to address the *data analysis* phase of the calibration procedure. These subVIs were collected under different *LabVIEW Libraries* to differentiate between their purpose. A set of *statistical analysis* subVIs was developed, including programs capable of generating a histogram from the feature columns of the calibration datasets and spotting the presence of spurious values, known as ***outliers***.

Another library, *Calibration.lvlib*, includes all the subVIs that are used by the main.vi to analyse the calibration datasets extensively. These subVI are capable of computing the expected value and standard deviation of each feature column and evaluating the ***standard uncertainty*** associated with the results found for the ascending and descending phase datasets collected during the acquisition procedure. Further, the **Hysteresis** phenomenon was addressed with a specific subVI which compares the maximum deviation between the ascending and descending phase expected values, with the combined uncertainty of the datasets. This procedure was followed for both sensors, and after that, their expected output values were compared to assess whether there exists some kind of relationship between the measures of the reference sensor and those of the DUT. The last step involved the estimation of a linear regressor that could be used to predict a flow rate measure comparable with that of the reference sensor, starting from the raw voltage output of the DUT. Such *calibration conversion equation*, has been computed for both the ascending and descending phase calibration datasets. Hence, depending on the input current phase, a different conversion equation can be used.

Concluding Remarks

The datasets collected during the testing phase of this project were analysed with these subVIs. The analysis revealed that **both sensors are affected by Hysteresis**, since the ascending versus descending curves maximum deviation, greatly exceeds the combined uncertainty computed. The comparison between the two sensors' output curves, during the ascending and descending phases, revealed that **there is a relationship between the two transducers' outputs**. This relationship has been estimated by linear regression, which in both the ascending and descending phase dataset resulted in an acceptable approximation. To quantify the measurement system overall uncertainty, and to provide a report of the expected values and standard uncertainties for each operating point, a **calibration report** was generated by the ***report.vi*** subVI, within the *Calibration Procedure Loop* in the main.vi.

We conclude that the automated measuring system developed is capable of completing each required task as required by the calibration procedure.

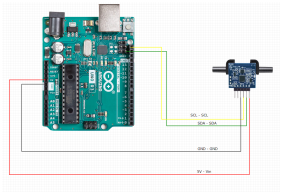
Chapter A

Arduino Script

Circuit

Pin FS-2012	Wire Colour	Pin Arduino
Vin	Red	5V
GND	Black	GND
SDA	Green	SDA
SCL	Yellow	SCL

(a) Arduino UNO - Reference Digital Sensor Pins Assignment



(b) Arduino UNO - Reference Digital Wiring

Figure 56: Overview of the Arduino UNO R3 - Digital Transducer Connection.

This Appendix details the Arduino script, found in the project files, responsible for acquiring digital sensor measurements after converting the transducer's output via its datasheet-specified function.

Library Inclusions

The sketch utilizes essential libraries: `Wire.h` for fundamental I²C communication, and `Arduino.h` for core Arduino functionalities. `I2Cdev.h` is included but not explicitly used in the current version.

Pin Definitions and Parameters

Hardware connections are defined by standard I²C pins on Arduino Uno: **SDA (A4)** for data and **SCL (A5)** for clock. The fixed I²C slave address for the FS2012 sensor is **FS2012_ADDRESS (0x07)**.

Global Variables

Key global variables include `data` (16-bit raw sensor output), `flow` (converted air flow value), and `delayMs` (delay between readings, initially 500 ms for 2 Hz sampling).

Sensor Reading Function: *digital_flow_sensor()*

This function encapsulates the I^2C protocol for reading the FS2012 sensor. It initiates a transmission to `FS2012_ADDRESS`, requests reading from register 1, and then retrieves two bytes. These bytes are combined (`a << 8 | b`) into a 16-bit `data` value, which is subsequently converted to L/min by dividing by 1000.0.

Initialization Function: `setup()`

The `setup()` function performs initializations upon Arduino startup. It sets up serial communication at 9600 baud using `Serial.begin(9600)`, initiates the I^2C bus with `Wire.begin()`, and includes a 500 ms delay for stabilization.

Continuous Execution Function: `loop()`

The `loop()` function runs continuously, managing the primary operational flow. It features serial command control to modify the sampling rate (e.g., via `SF:<frequency>`), calls `digital_flow_sensor()` to update the `flow` value, incorporates a `delayMs` pause, and prints the measured `flow` value to the serial monitor.

Serial Event Handler Function: *serialEvent(String command)*

This function, invoked when serial data is available, parses and acts upon incoming commands. It trims the `command` string, checks for the "SF:" prefix, extracts the frequency value, converts it to a float, and, if positive, calculates and updates `delayMs` (e.g., `delayMs = (unsigned long)(1000.0 / newFreq);`). This mechanism enables dynamic adjustment of the sensor's sampling rate via serial commands.