POLITECNICO DI TORINO

Master's Thesis in Mechatronic Engineering

Test bench for pneumatic cylinders: control, data acquisition and HMI with low cost programmable controllers

Supervisor
Prof. Luigi Mazza
Co-supervisors
Prof. Andrea Manuello Bertetto
Eng. Marco Pontin

Candidate
Michea Mezzasalma
232756

A.Y. 2018/2019

ABSTRACT

The large diffusion of pneumatics in the field of automation leads to the necessity of devices that may guarantee high reliability. Consequently, it is essential to establish and evaluate the performance of pneumatic actuators with regard to the service life, influenced by the proper functioning of constitutive parts such as the bearings or the gaskets, which critically affect the efficiency of the cylinders.

Therefore, the goal of this thesis is the design and implementation of a test bench that aims to assess the reliability of pneumatic actuators through some tests, which provide information about the pressure of the compressed air inside the chambers and thus the sealing capability of the gaskets over time, affected by the exposure to wear phenomena. These tests finally allow the evaluation of the lifetime of the actuators. The test bench has been designed to operate in a fully automatic way, allowing at the same time the interaction of users by means of pushbuttons.

Instead of the expensive PLC, traditionally used in previous test benches realized at the Mechanical Department of Politecnico di Torino, our test bench features only low cost programmable controllers such as Arduino, which serves as main controller and data acquisition system, and Raspberry Pi. In particular, this last board is used to override memory problems and store the results coming from the test bench. To check these results, as Human Machine Interface (HMI) users can employ not only a monitor directly linked to the Pi board but also a PC connected to the same network of the Raspberry Pi, thanks to the Web Server hosted by the board.

Among the goals set at the beginning of the work, there is also the realization of a code in Arduino which may be well-structured, adaptable and expandable in case of future developments for the test bench, such as the insertion of additional pneumatic cylinders.

Table of Contents

1 INTRODUCTION

Pneumatic cylinders are mechanical instruments that employ the power of compressed air in order to generate a force in a reciprocating linear motion. The part of the cylinder that performs most of the work is the piston, a moving element attached to a piston rod. The movement of the piston takes place inside a cylindrical bore. The bore or diameter of a cylinder determines the highest force that can be produced by the device, while the maximum linear movement that can be generated by the cylinder is determined by the stroke. The force that a piston can exert is affected not only by the piston's area but also by the pressure of the air supply ($F = P \cdot A$). For example, a 4cm bore cylinder working at 8 bar could easily lift a 100kg man.

Air cylinders are widespread devices nowadays, and this is due to the fact that not only they are strong enough to resist harsh environments, but they are also capable of providing high performances at low cost, good power density (ratio between power and dimensions) and simple and easy maintenance. All these characteristics make the air cylinders strongly desired instruments in many industrial applications, such as lifting and moving merchandise, removing things from conveyor belts and putting things on conveyor belts, opening and closing valves and doors and so on. Air cylinders have also been employed in robotics for robotic arms, as well as in mechanical, space and medical applications. In particular, when applications involve high speed, linear motion, and moderate loads, pneumatic cylinders are usually the first choice to provide the actuation.

The reason behind the use of pneumatics, or any kind of machine where energy is transmitted, is to execute work. Work is achieved when kinetic energy is applied on a load that needs to be moved over a certain distance. As far as pneumatics is concerned, this kinetic energy is obtained from the conversion of the potential energy stored inside the compressed air. When the compressed air accumulated inside a tank is released by

opening a valve, which is needed to control the air flow, the air expands trying to reach the atmospheric pressure and so in this way it can perform a work.

When the valve is opened, the compressed air coming from the tank enters the duct at one end of the air actuator, applying a force to the piston and consequently to the piston rod connected to it. The air expands inside the two chambers of the cylinder, leading to the movement of the piston. The cylinder contains some internal seals that have the purpose to keep the air inside the two separated chambers, positioned on each side of the piston. As the piston rod moves, the air accumulated in the other chamber during a previous cycle has to be exhausted.

When a load, on which some work has to be performed, is mounted on the piston rod there are some situations that must be avoided, for example the mounting of the load on the side of the piston rod, because the piston rod is designed for an extend-retract movement. Moreover, a heavy load mounted perpendicularly to the axis of the piston rod leads to the bending of the rod, that can damage the front end seals or internal seals of the cylinder.

However, even if these situations are avoided and the cylinder operates under normal conditions, after a certain period of functioning pneumatic actuators will present eventually internal or external air leakages. Internal leakages between the two chambers are often due to the wear of the internal piston seal, but can also be traced to gouges or dents on the piston rod. In both cases, the substitution of these elements is necessary. External leakages instead are present in the rod seal and wiper located at the front end of the cylinder, that has two functions: pressure sealing, with the purpose of preventing the leak of compressed air from the space between the piston rod and the front end bearing, when the front chamber is under pressure; wiping, since the piston rod is cleaned every time it reenters the cylinder, in order to prevent particles from being drawn back inside and consequently reducing the life of the internal seals and the bearing.

The wear of the bearing, caused by loads that repeatedly provoke sliding contact fatigue, grows at each cycle until the bearing can no longer keep the alignment of the piston rod

with the axis of motion. The piston rod starts to lean on the seal, wearing it out progressively until the cylinder stops working, since the compressed air is no longer kept properly inside the front chamber. In this case, the force exerted by the compressed air is heavily reduced so that it becomes insufficient to move the loads that it was intended to push or, in a worse situation, the piston does not move at all even when no load is present.

The life of a pneumatic cylinder can be defined as the total distance covered by the piston before the pressure of the air inside one of the cylinder chambers becomes lower than the 5% of the supply pressure, under some conditions. These conditions consist in the fact that the pressure has to be checked for both chambers, with the piston blocked for a predefined amount of time; if during this time the pressure becomes lower than threshold value, we can consider the cylinder broken. We decided to evaluate the duration of a cylinder in this way, even though there are other possible ways, for example considering the air leakages by means of a flux-meter and leak tests. Life can vary because wear in the cylinder is provoked by a statistical phenomenon, that is material fatigue.

We built a test bench that, by taking into account what happens to the pressure inside the chambers, evaluates the duration of the seals, so that when the gaskets are not able to fulfill the task for which they were designed and the losses are unsustainable, it means that the actuator reached the end of its life and it has to be regenerated. The sealing capability of the gaskets is considered a quality index for the cylinder. For our test bench, the phase in which we check the pressure values blocking the piston for a certain time is called “seal test”.

During what we have called, instead, “wear test”, which is the phase in which the cylinder has to complete a predefined number of cycles, the objects are intrinsically consumed and exposed to the wear phenomenon. We know that to evaluate the sealing capability over time, the piston rod has to slide over the gaskets. However, the test bench has not been designed to produce measurements on volumes, consumption and removed material. We are aware of the fact that in literature there are many mathematical models regarding the wear phenomenon and the different wear types

(abrasive, adhesive etc.) studied by tribology. We also know that there are experimental tests conducted with the employment of instruments such as tribometers, that can measure volumes of removed material when two surfaces are in contact. Our goal, anyway, was just to evaluate the performances of the cylinders in terms of efficiency of the sealing capability of the gaskets, recognizing their malfunctioning when they become unable to keep the air in the chambers.

The test bench is physically composed by many pneumatic and electronic parts. We will consider all of them in the following chapters, starting from the pneumatic network. In Figure 1-1 we show a front view of the test bench. In the picture, it is possible to recognize the pneumatic cylinders (only two of them are actually used), the pressure sensors, the Raspberry Pi board and a box. The box contains most of the electronic network, while on its cover it is possible to find buttons and LEDs that allow the interaction of the user with the test bench.

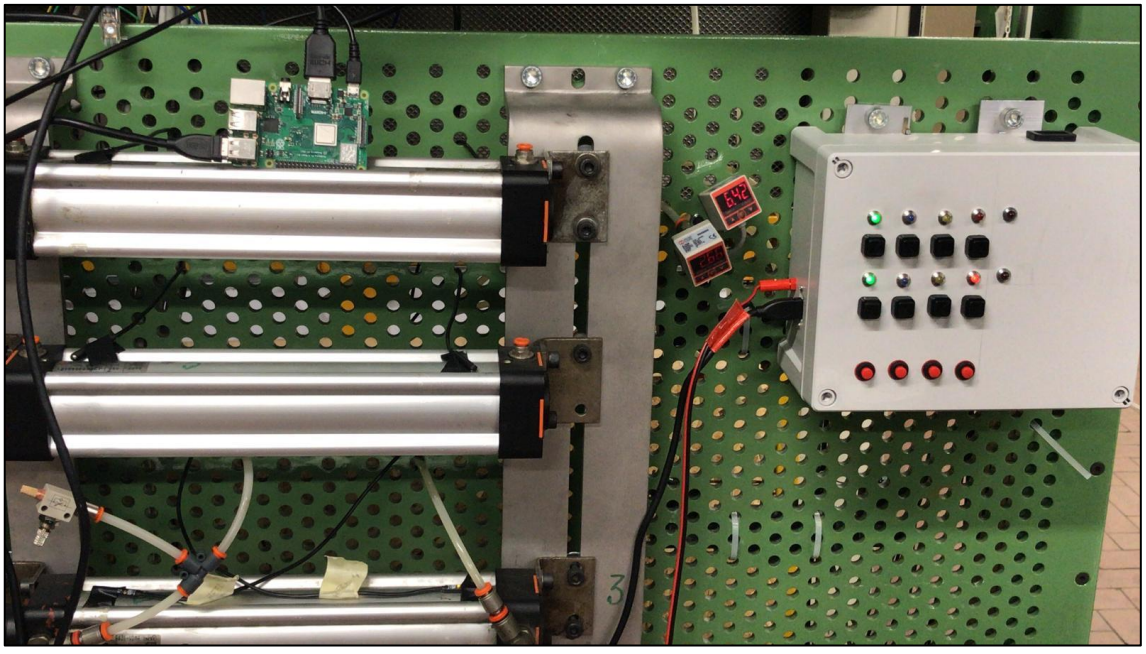

Figure 1-1 Front view of the test bench

2 PNEUMATIC NETWORK

The pneumatic network is composed by two piston cylinders, then for each piston we have a 5/3-way valve, an OR valve, a pressure switch, a pressure transducer.

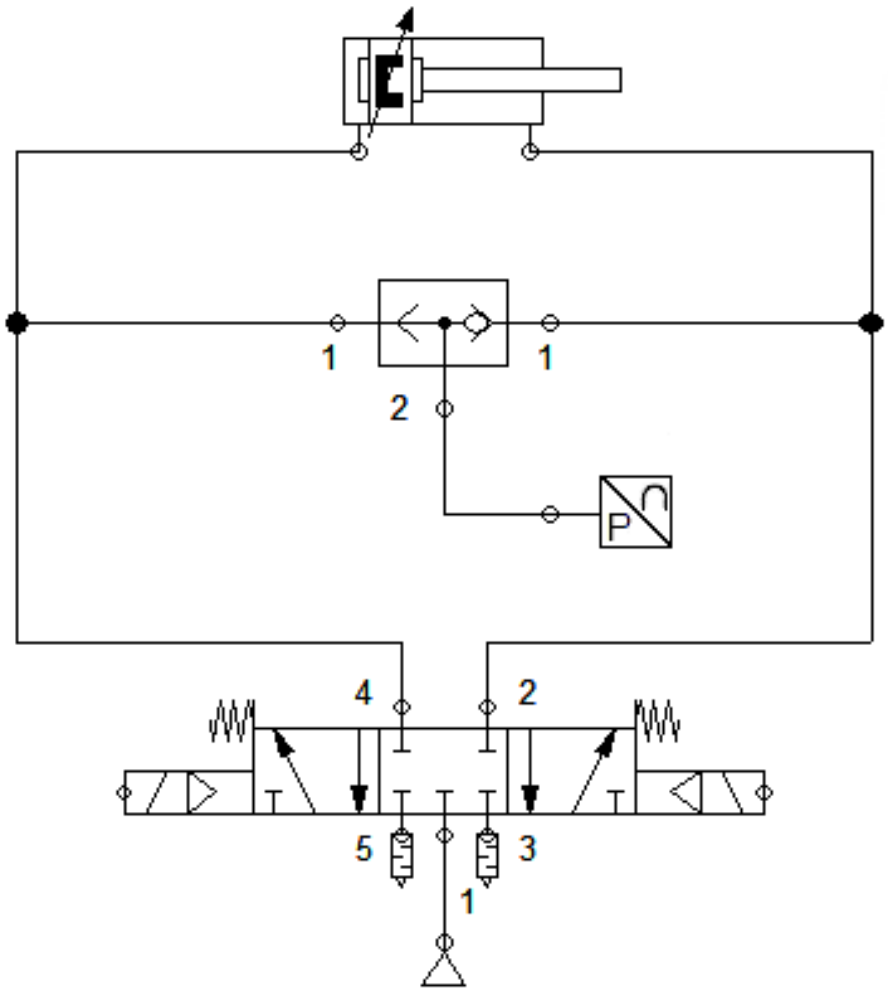

Figure 2-1 Pneumatic network schematic

In the pneumatic circuit pressure sensors have been inserted, one for each cylinder. Pressure sensors provide two functionalities: pressure switches (digital output) and pressure transducers (analog output). The pressure sensors have been connected to the

rest of the circuit by means of the logic OR valves: in this way, it is possible to read the pressure independently of the particular chamber of the cylinder. The inputs to the Arduino board are reduced by half because we only have one input pin for each piston dedicated to the reading of the pressure of both chambers. In particular, at the moment in our test bench we only use the pressure transducers.

The logic OR valve acts as a flow selector, guaranteeing that the pressure transducer is in contact with one chamber at a time, the one where the pressure is higher and on which we are performing the seal test. This valve is basically a ball valve with two inputs (connected to the chambers of the cylinder) and one output (connected to the pressure transducer). The internal sealing disc is pushed by the compressed air entering from each of the inputs alternately, so that it seals the other signal line to prevent loss of pressure, as shown in Figure 2-2.

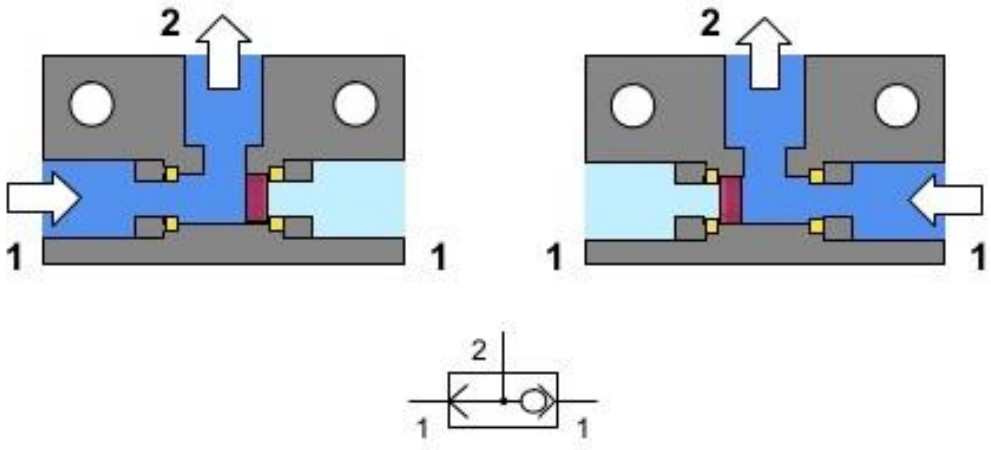

Figure 2-2 Logic OR valve [1]

The piston cylinders used in the test bench belong to the brand MetalWork with reference ISO 15552. They feature 50 mm bore and 250 mm stroke. They are double-acting components: they have two ports for letting the air in and out, one for the front chamber and one for the rear chamber, and can perform work in both directions. Moreover, these actuators are magnetic cylinders that have a band of magnetic material inserted around the piston circumference, polarized in the same direction of the axis of

the cylinder. The barrel, instead, is made of non-ferrous material. The presence of magnetic material is what allows the use of Reed sensors, that will be discussed later on [2].

The pressure of the air in the cylinders is limited to a maximum value of 10 bar thanks to a pressure regulator inserted between the compressed air supply, already provided at a pressure of 10 bars, and the rest of the pneumatic circuit. We inserted the regulator in order to have the opportunity to perform seal tests on the cylinders not only at 10 bars but also at lower values of pressure inside the chambers.

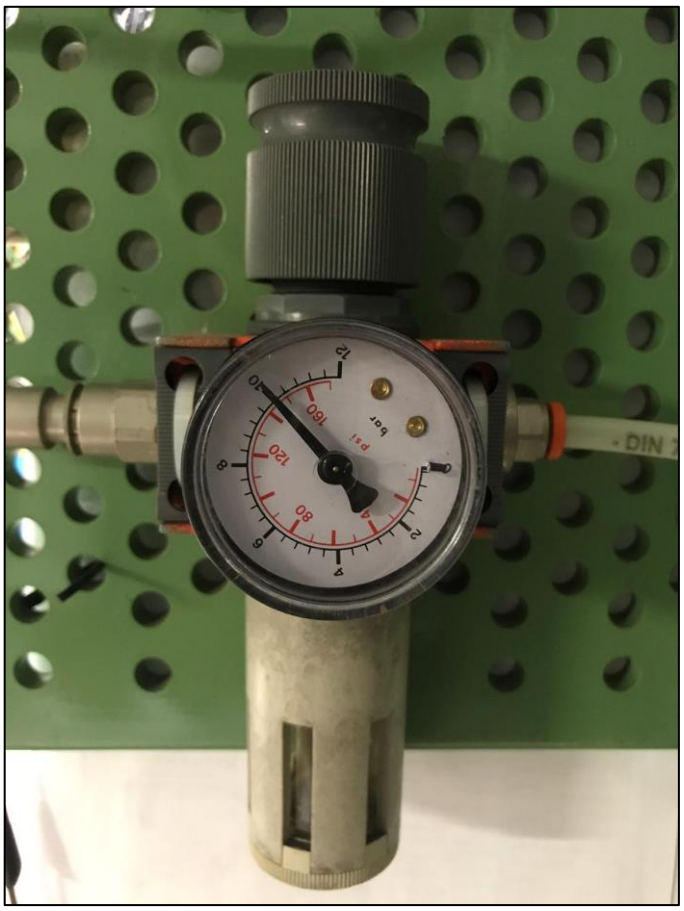

Figure 2-3: Pressure regulator

2.1 DIRECTIONAL CONTROL VALVES

The Wairkom 5/3-way valves SUK SUK used in the test bench are directional-control valves. The number “5” inside the name refers to the fact that the valve has five working ports. The five ports are: one supply, two air actuator ports and two exhaust port. The number “3” in the valve designation identifies the number of the valve positions.

The 5/3 valve internal structure makes it so that compressed air can flow into one port of a double-acting cylinder and at the same time exhaust from the other port on the same cylinder. By shifting the internal flow paths of the valve, the 5/3 air valve sends compressed air alternatively to each of the two ports of the actuator, receiving the exhausting air from the other port, allowing in this way the functioning of the double acting air actuator.

In addition to the flow commutating function that allows the normal operation of the cylinders (left and right positions, unstable), the valve can also provide a blocking position (central position, stable), useful during the seal test. In this position all the ports of the valve (supply, actuator and exhaust) are blocked. No air can flow through the valve to either actuator port as the supply path to those ports is blocked. Air also cannot travel from either actuator port to either exhaust port, since those paths are closed too. In this position, air cannot flow through the valve to the cylinder or from the cylinder back through the valve, then when the valve shifts into “Blocked Center”, the air actuator will stop dead. Our intent when we chose a “Blocked Center” 5/3 valve was that when the valve is “at rest” we want the cylinder to be frozen. Air cannot enter into or exit from the cylinder, which freezes. The circuit is closed, so that in this way every spot of the cylinder, which could be sensitive to losses of compressed air, is eliminated. It is, indeed, easy to understand that any undesired loss can provoke a pressure drop that, when read by the pressure transducer, can compromise the seal test. In a similar case the pressure drop could be wrongly linked to the gaskets of the cylinders, while they could actually be entirely undamaged.

The 5/3-way valves are electro-pneumatic driven valves. They are actually composed by two internal types of valve: a pilot valve, consisting in a direct acting solenoid valve used

as a control element, and a pneumatically actuated valve having proper dimensions (bigger than the pilot valve) used as an amplifier element. This is done because small solenoids are not sufficient to operate with high flow capacities. In order to excite the solenoid and have the commutation of the valve, a minimum energy is required; then the passage of the compressed air is allowed and the commutation of the bigger valve takes place.

Inside the bigger valve there is a sliding spoon, as shown in Figure 2-4. When the spool is in the central position all the ports are sealed (Figure 2-4 a); when the spool is on the left port 1 is joined to port 4 and port 2 is joined to port 3 (Figure 2-4 b); when the spool is on the right port 1 is joined to port 2 and port 4 is joined to port 5 (Figure 2-4 c) [1].

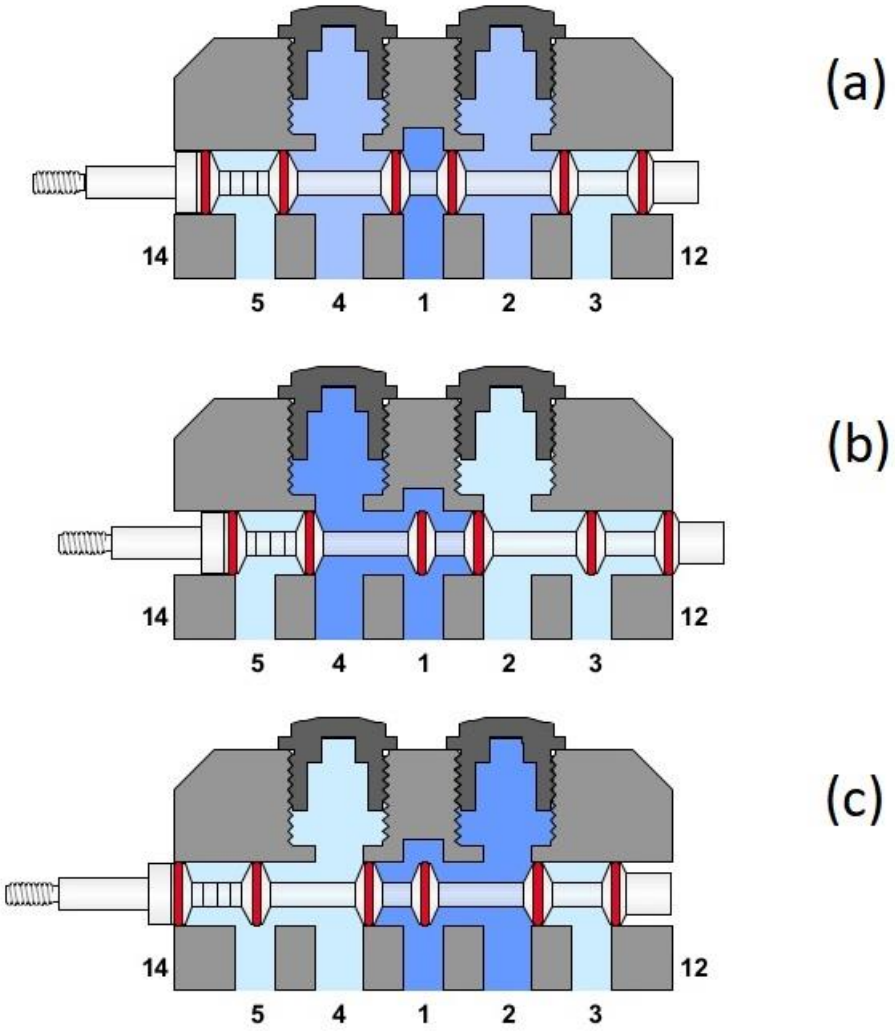

Figure 2-4 Three positions for the spool valve [1]

In Figure 2-4, port 1 is the supply port, port 2 and port 4 are the actuator ports, port 3 and port 5 are the exhaust ports.

The 5/3-way valves (along with the pressure transducers) are the elements that connect the pneumatic circuit of the test bench to the rest of the system. Their behavior is controlled by the Arduino board through a series of driving electronic components. In the following section we will consider and describe the electronic part of the test bench.

3 ELECTRONIC NETWORK

The electronic network is the part of the test bench that makes possible for Arduino to control the rest of the system. It is composed by many different parts that have to be examined one by one.

The core of the electronic network is a veroboard on which we welded a lot of electronic components; these components are linked through a series of terminal block connectors and wires to another smaller veroboard mounted on the Arduino Mega 2560 and connected to it by means of pin headers. All these just mentioned elements are inserted inside a box, in order to keep them safe and avoid that a wire could be easily disconnected. Figure 3-1 shows a view of what can be found in the box.

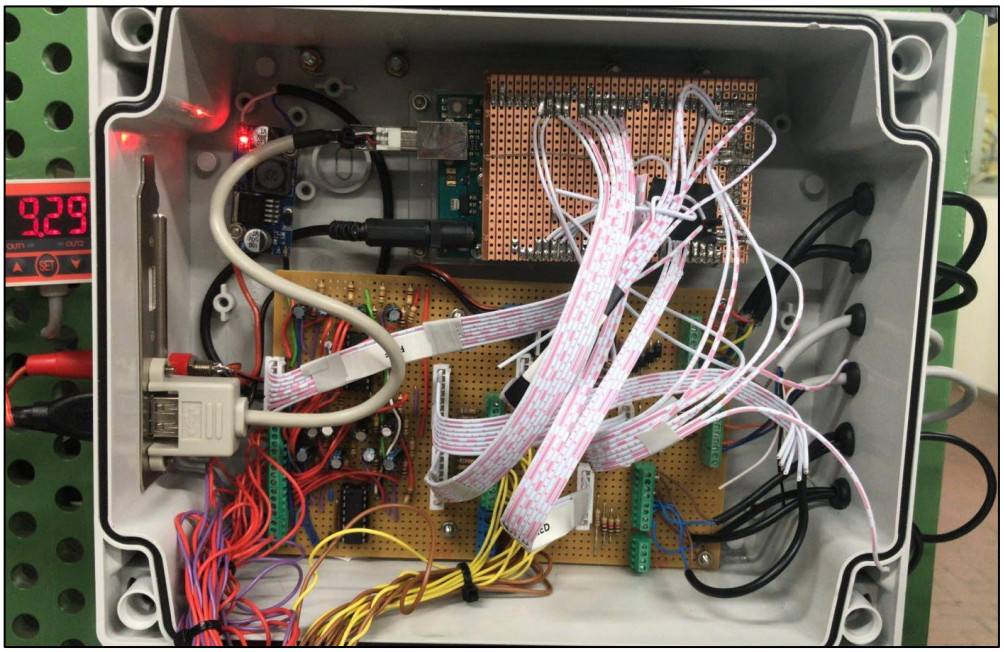

Figure 3-1: Internal view of the box

Wires also connect the main veroboard to the buttons and LEDs that are placed on the external part of the cover of the box, where the interaction of the user with the system

takes place. Inside the box we also installed a resistive trimmer, that is connected to a 24 Vdc power supply unit and it is set to provide a constant output voltage of 5 volts.

The presence of the trimmer is due to the fact that the required power supply is not the same for all the parts composing the system. In particular the 5/3-way valves and the pressure sensors require a 24 volts voltage supply, while the rest of the circuit needs 5 volts in order to work.

3.1 HARDWARE DEBOUNCING OF THE BUTTONS

Using pushbuttons means having to deal with one of their issues called “switch contact bounce”. When a button is pressed, there is not just a simple passage from open circuit to short circuit. Inside the pushbutton there are two pieces of metal that have to come into contact to make the conduction happen. These two pieces can make and break contact multiple times before the conduction between the two of them is stable, so it may appear to the microcontroller, that can test the state of the button at a very high frequency, that the button has been pressed many times in a very short time, while the user actually pressed it only once. In order to avoid this misunderstanding between the user and the microcontroller about when a button release or push event happened, it’s necessary to perform what is called the “debouncing” of the buttons.

Two solutions are possible to fix the problem of switch bounce: hardware debounce and software debounce. Considering the fact that the software solution could be much harder to implement than the other one that just involves the use of few electronic components, we chose to adopt the hardware solution.

Fast switching in a short time means that in the frequency domain the voltage shows many high frequency components. We needed to remove them to eliminate the wiggles in the time domain, so for this reason we needed a low pass filtering action. The simplest solution was to add a resistor and a capacitor using an RC filter. We replicated the scheme [3] shown in the following figure:

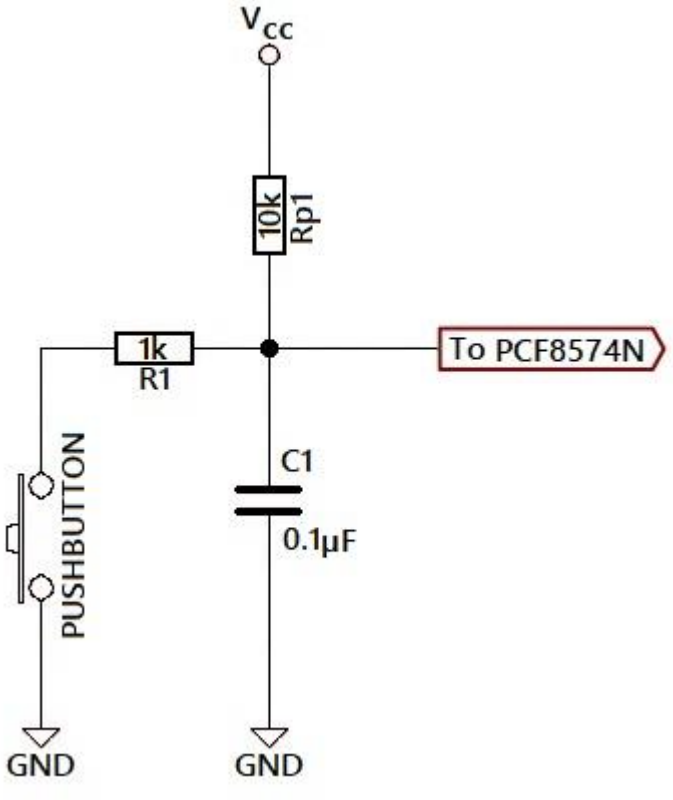

Figure 3-2: Hardware debouncing of the pushbuttons

We chose to adopt this solution as it was good in terms of money and space occupied on the veroboard, and it also revealed sufficient for our purposes. The values of R and C are chosen so that the product R*C is approximately near the time we would like to debounce for. We used a 10 Kohm resistor and a 100nF capacitor, so the product is 0.001 seconds (a millisecond). We also had to add a 1K resistor for the switch, that has to be significantly lower than R so that the switch can settle on a low voltage when the button is pressed. This was necessary in order to avoid the dumping of the capacitor directly through the button, which could lead to the appearance of undesired high-frequency voltage noise.

As it is possible to see in the Figure 3-2, the output voltage is high when the button is released, low when the switch is pressed. This output voltage is used as an input for the pins of the PCF8574N, the element that we are going to describe in the next paragraph.

3.2 PCF8574N

In order to connect the external buttons to the Arduino board, we employed a particular kind of integrated circuit called PCF8574N (we are also going to refer to it as “PCF” for the sake of brevity). This device is an I/O expander, useful to grant more input/output pins (8 pins for each PCF) than the ones provided by the Arduino Mega. The nomenclature of all the pins of the PCF8574N is shown in the following figure:

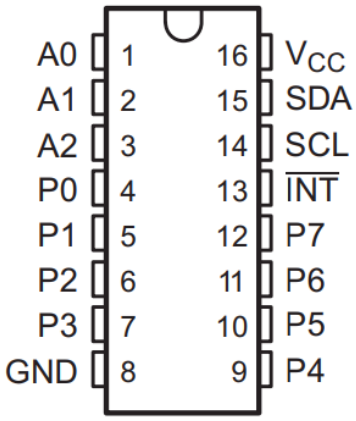

Figure 3-3 Pins of PCF8574N

The PCF8574N communicates with the Arduino board by means of the two-wire bidirectional I2C-bus (serial clock (SCL), serial data (SDA)) and is designed for 2.5-V to 5-V V_{CC} operation [4]. In our case, we decided to feed the I/O expander employing the 3.3-V voltage supply provided by the Arduino board as an output, instead of 5 Volts as in the case of the rest of the circuitry powered by Arduino. This was done in order to prevent overheating phenomena that could badly damage the integrated circuit.

Having to connect each external button to a pin of the PFC8574N meant that one integrated circuit was not sufficient, so we mounted two I/O expanders by using proper chip sockets. We made use of all eight input pins of one PCF (we will refer to it as “PCF1”), and for the remaining four buttons we used P0, P1, P7 and P8 of the other PCF (“PCF2”).

The working principle of the PFC8574N is the following: whenever an input pin changes its state, an interrupt is generated. Each of the two PCFs provides an active LOW open-

drain interrupt output ($\overline{\text{INT}}$) which is connected to the Arduino microcontroller, that has to read the state of the PCF that sent the interrupt in order to detect which pin of the PCF has gone through a change of state, to find out in this way which button was pressed. The voltage of the interrupt output pin $\overline{\text{INT}}$ is normally high and becomes low when the interrupt is generated, so the pin is connected to 5V by means of a 10 kΩ resistor.

We are using Arduino pins 53 and 52, on which pin change interrupts are enabled (Port B). If we had avoided to employ PCFs, on Arduino we should have used 12 pins where pin change interrupts are available, instead of just two. Considering the fact the Arduino Mega provides only 24 pins (3 ports) for interrupts on change (and some of them had to be used for the Reed sensors), we saved some resources on the Arduino thanks to the I/O expanders (this would have been absolutely necessary if more than two cylinders had been present on our test bench).

To allow the communication among the two PCFs and Arduino, we had to connect the SCL and SDA pins of PCF1 to the Arduino digital pins 21 and digital pin 20 respectively on one side, and to the SCL and SDA pins of PCF2 on the other side. Moreover, we had to specify an address for the two PCFs by using their three address pins. For PCF1 we chose as address “000” (A2 = A1 = A0 = 0), while the address “001” was chosen for PCF2 (A2 = A1 = 0 and A0 = 1). In these addresses, “0” means connecting to low voltage level, while “1” corresponds to high voltage level.

3.3 DRIVING CIRCUIT FOR THE 5/3-WAY VALVES

In order to drive the 5/3-way valves, some power electronic circuits had to be added on our main veroboard. This is necessary because the valves are fed with 24 V, and their power absorption is around 5 W. For our discussion, in this paragraph we will just consider the internal solenoid of the valve, which will be modeled as an inductor.

In our test bench we have two 5/3-way valves, and each one of them contains two solenoids, so we had to supply power to four solenoids. The circuit that we implemented [5] for each solenoid and we had to replicate four times on the veroboard is the following one:

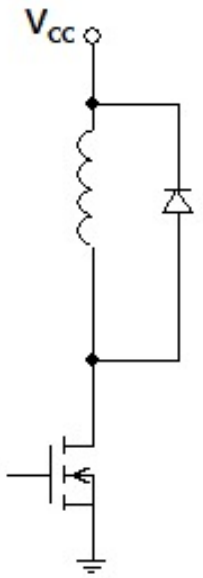

Figure 3-4: Driving circuit for the valve solenoids

The MOSFET we used is an IRF520N, which is a very widespread power mosfet. It is an N-CHANNEL enhancement type mosfet: this means that the passage of current between the drain and source pins only happens when a proper voltage is applied on the gate pin of the mosfet [6]. Using a simplified model, we can consider the mosfet as a switch: the transistor acts as a short circuit when the voltage on the gate is high; it behaves as an open circuit when the voltage on the gate is low. In this configuration, the power dissipated on the transistor (given by the product of the current and the voltage across it) is always minimal, and all the energy from the power supply is used indeed by the solenoid.

The gate of each mosfet is connected to a different pin of Arduino, which in this way can control the mosfets by applying a voltage level of 0 or 5 volts. We are in the case of simple ON/OFF driving.

Between the output of Arduino and the gate of the mosfet, a 1 KOhm resistor has been inserted in series to limit the current on the output during the commutation of the mosfet.

3.3.1 THE USE OF THE PROTECTION DIODE

We will now consider the reason why diodes were employed in the circuit. As we said, the solenoid acts as an inductor, a component that tends to keep constant the current across its terminals. When the transistor in Figure 3.4 is turned on, the current reaches its steady state value after a certain time, following an exponential curve that depends on the ratio between La and Ra of the equivalent circuit. To a first approximation, this does not cause any trouble.

When a transistor is turned off, the current across it should instantaneously reach the zero value. The inductor, however, tends to block this sudden decrease of the current and for this reason provokes an increase of the voltage on the collector pin of the transistor (we can imagine that in this phase the transistor becomes suddenly a very high value resistor where the inductor is trying to drive some current: for Ohm's law, the voltage has to increase). The voltage easily gets to a value of hundreds of volts, damaging the transistor itself. This voltage is what is usually called "flyback voltage".

In order to prevent this destructive phenomenon, it is inserted in parallel to the solenoid a diode, that guarantees to the current an alternative way with respect to the one provided by the transistor when this one is opened.

The cathode of the diode is connected to the voltage supply (in our case 24 volts). In figure 3.5 it is represented the path followed by the current: when the transistor is on, the current flows through the solenoid and the transistor, while the diode is off, being reverse-biased; when the transistor is off, the same current that passed across it now flows through the diode. Obviously, this last situation endures for a very short time, given the fact that there is no generator being able to keep constant the passage of the current.

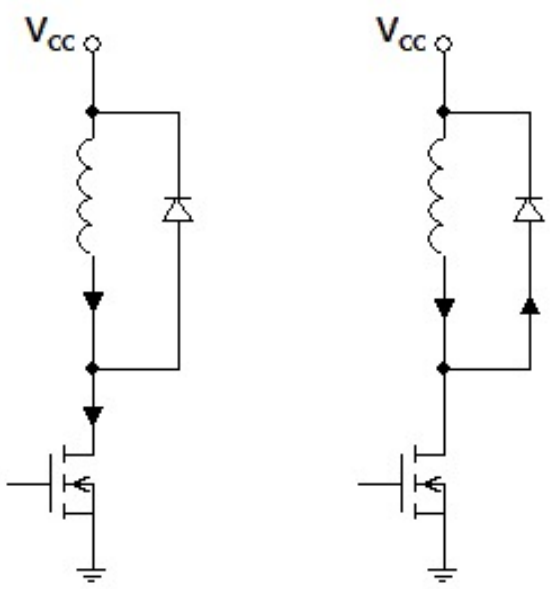

Figure 3-5: Paths of the current for the driving circuit of the valve solenoid

In our application, we needed to use diodes showing two important characteristics: being fast, that means being able to switch easily from a condition where there is no passage of current to a condition of conduction and vice versa; being able to deal with high currents, because during the turn-off the current from the solenoid goes directly into the diode. The 1N4007 that we used proved to be good enough for the task.

3.4 REED SENSORS

In order to detect all the times that the piston rod reaches the end of the stroke inside the cylinder and to provide this information to Arduino, Reed sensors had to be applied at both sides of each cylinder in the test bench. Considering the fact that we have two piston cylinders in our test bench, we had to use four Reed switches. These devices are magnetic proximity sensors, meaning that they are non-contact proximity devices that are employed to detect magnetic objects. In our case, they monitor the piston position, recognizing the field of the magnet integrated into the piston through the cylinder wall. This magnetic field can induce magnetic poles into the metal parts of the reed switch

and the consequent attraction between the electrical contacts causes the activation of the reed switch. [7]

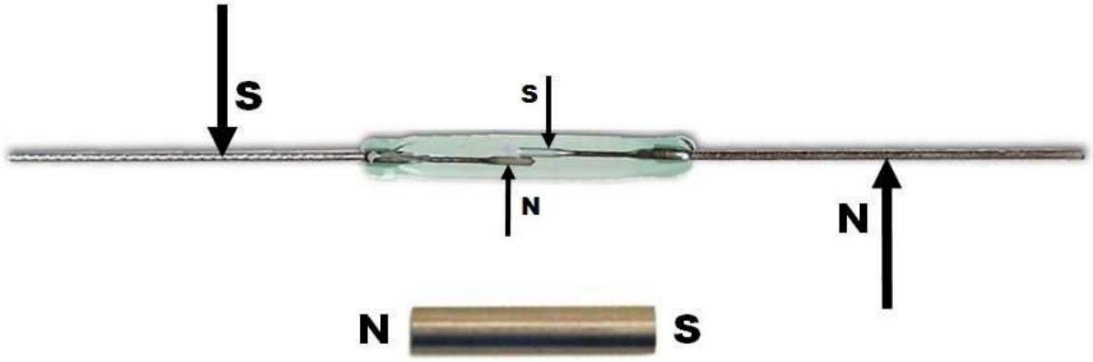

Figure 3-6: Reed contacts and magnetic induction [7]

The attraction between the electrical contacts completes an electrical circuit, turning on the internal LED of the sensor.

Reed sensors can have two or three external wires. In a first designing phase, we had planned to use three wire sensors, so we placed terminal block connectors and resistors having that goal in mind. The three wires of these sensors are: brown wire, that has to be connected to the supply (5 volts, provided by Arduino); blue wire, that has to be connected to the ground; black wire, that returns the output signal (high when the switch is turned on, low when the switch is low) and has to be connected to an Arduino pin. Between the black and the blue wire we inserted as a load a 1 kOhm resistor.

In the end, according to what was available in our laboratory, we used two wire sensors. They have a brown wire (to 5 volts) and a blue wire that returns the output signal. It was sufficient to connect the blue wires inside the terminal blocks that were destined to the black wires of the three wire sensors.

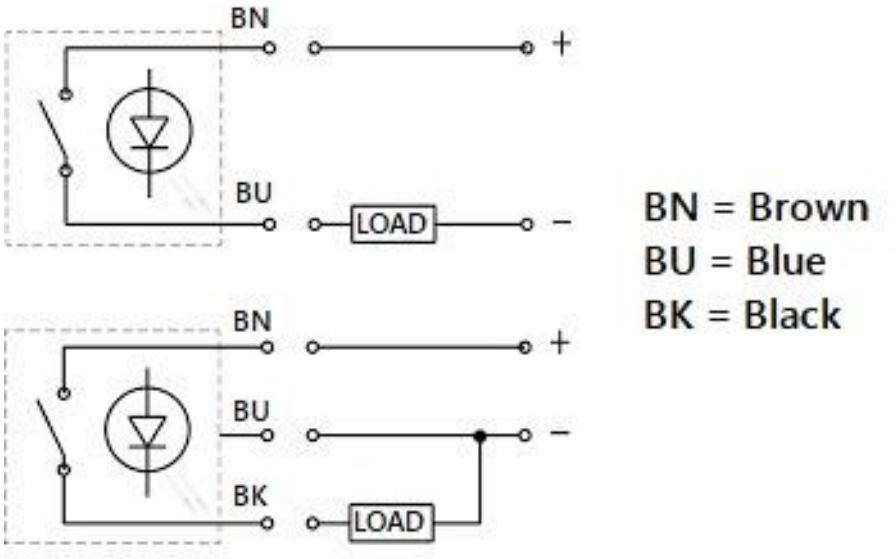

Figure 3-7: Wires for two-wire and three-wire Reed sensors

Reed sensors were connected to the Arduino pins from 64 to 67.

3.5 RESISTORS FOR THE LEDS

In our test bench we have ten LEDs placed on the cover of our box. They are connected to the Arduino Mega board, in particular to the pins numbered from 2 to 11. The 5 Volts voltage supply is provided to the LEDs by the same board.

For each LED we had to add on the main veroboard a resistor to limit the current on the LEDs, with the purpose of avoiding to burn them. We used 6.8 kOhm resistors for every LED, noticing that with these resistors the LEDs showed a proper brightness.

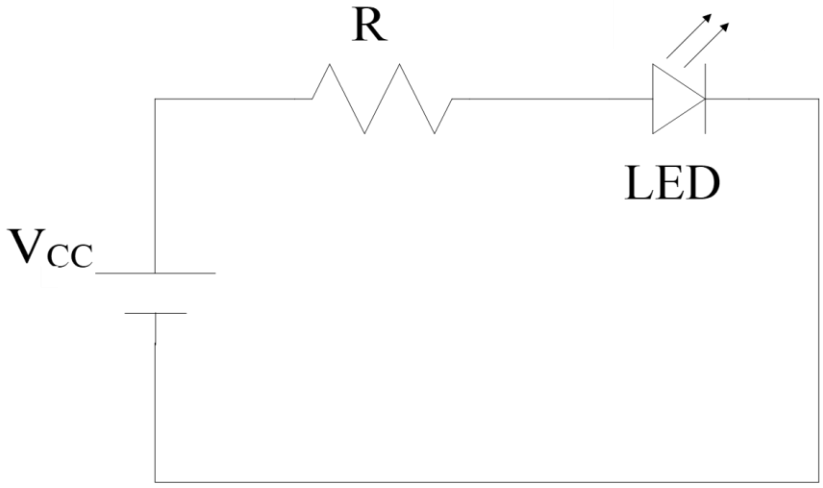

Figure 3-8: LED circuit

3.6 CONNECTION OF THE PRESSURE SENSORS

The pressure sensors feature five different wires: brown, that has to be connected to the 24 V voltage supply; blue, connected to ground; black and white, corresponding to the digital outputs of the pressure switch inside the pressure sensor; orange, corresponding to the analog output of the pressure transducer inside the pressure sensor. In our project we are just using the analog output of the two pressure transducers, connecting them to the pins 54 and 55 of Arduino. The analog signal provided by the pressure transducers can range from 1 to 5 Volts, so it was possible to connect the analog output directly to Arduino.

Even though the digital outputs are disconnected and no function that takes them into consideration has been implemented in the Arduino code, we placed some circuitry on the veroboard so that the pressure switches can actually be used in future developments of the test bench. Specifically, for each digital output we added a resistive divider as shown in Figure 3-9.

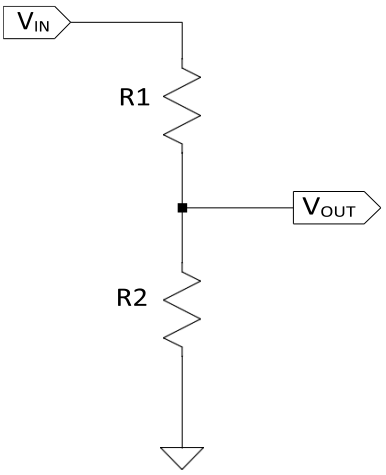

Figura 3-9: Resistive divider

We had to satisfy the following equation, valid for every resistive divider:

$$\frac{V_{OUT}}{V_{IN}} = \frac{R_2}{R_1+R_2} \qquad \text{Eq. 1}$$

In our case, V_{IN} is the maximum voltage for the digital outputs and it is equal to the voltage supply of the pressure switches, 24 Volts. V_{OUT} is the maximum voltage that we need to provide to Arduino: 5 Volts. The reasonable values that we chose to use for the resistors are R_1=8.2 kΩ, R_2=2.2 kΩ. The equation 1 is satisfied.

4 WORKING PRINCIPLE OF THE TEST BENCH

In this chapter, we will discuss about the main functionalities of the test bench and the basics of how the test bench works. We needed a system that could perform wear tests and seal tests on a specified number of cylinders (in our case, two) having the possibility to deal with each cylinder independently of what happens to the other air actuators. The structure that we gave to the whole test bench makes it so that while the system is working, each cylinder can be in his own independent state. In particular, we will now consider five fundamental states, related to the main situations in which each cylinder can be. It is important to know the fact that the decision to take into account five states is just an arbitrary assumption due to the need of a simplified discussion. By looking at the Arduino code, it is possible to realize that in fact there are also other minor states contained inside the main ones or connecting some of the five fundamental states between each other. These minor states will be mentioned in some cases in this chapter and will be examined better later on.

4.1 BUTTONS AND LEDS

Before going into detail about the states, it is necessary to know some information about what is on the cover of the box that is part of the test bench. An image of this cover is shown in Figure 4.1 in the following page. Each element on the cover is associated to a number. Then, in order to know the name we gave to each of these elements, we can consider the position of that number inside the Table 4-1. For example, number 1 corresponds to “first cylinder SELECT LED”, while number 20 is the “global WEAR button”. In Figure 4-1 we can see that there is a total of twelve buttons and ten LEDs.

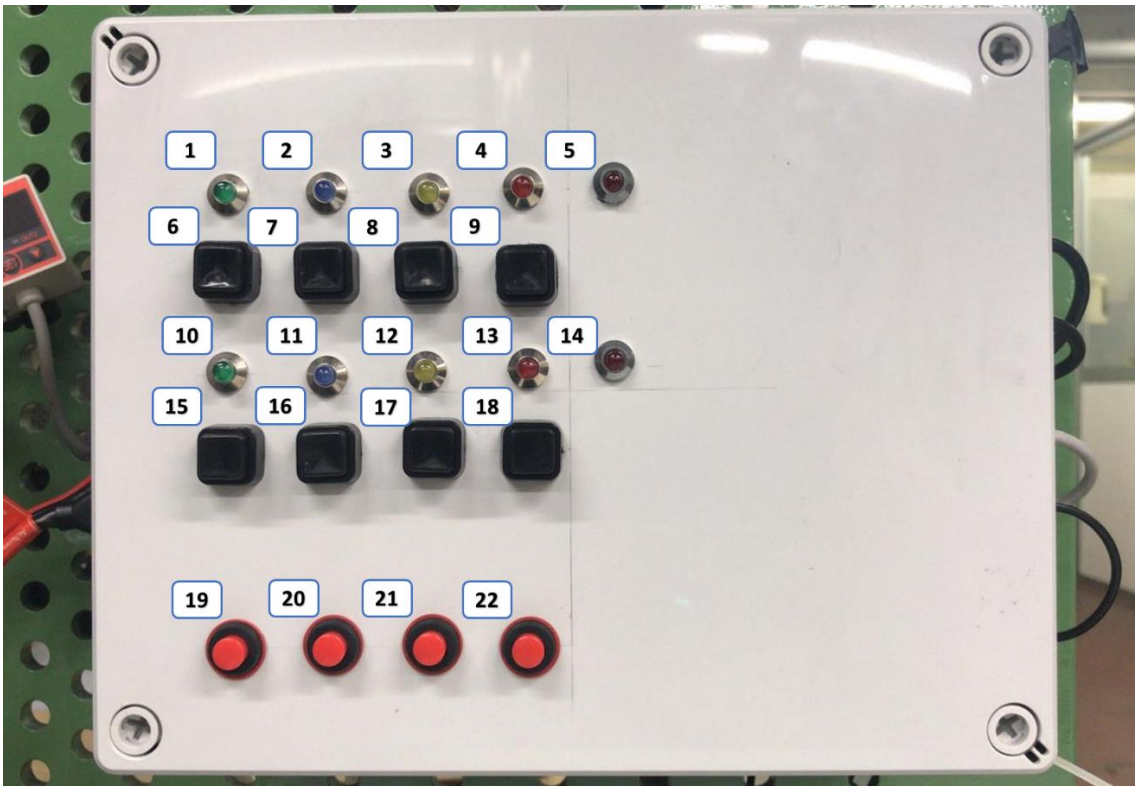

Figure 4-1 Cover of the box

		SELECT	WEAR	SEAL	STOP	BROKEN
1st Cylinder	LED	1	2	3	4	5
	Button	6	7	8	9	
2nd Cylinder	LED	10	11	12	13	14
	Button	15	16	17	18	
Global	Button	19	20	21	22	

Table 4-4-1: Elements on the cover of the box

We dedicated four buttons and five LEDs to each cylinder, and the functionalities that these components show for one cylinder are exactly the same for the other one. The remaining four buttons act on all the cylinders simultaneously, performing the same operations of the equivalent buttons related to the single cylinders. When a button is pressed but the related operation cannot be carried out on a particular cylinder in that

particular moment, the event is simply discharged and nothing happens. This case is more frequent when the global buttons are pushed, because there might be a situation when an operation is possible for one cylinder and impossible for the other one.

Buttons are often responsible for the passage of a cylinder from one state to another, but sometimes these passages may happen automatically because of some other events. The LEDs, instead, provide information about the state a cylinder is in.

4.2 FIVE MAIN STATES FOR THE CYLINDERS

In the following figure, we will examine a flow chart regarding the five main states mentioned at the beginning of the chapter. These states will be examined one by one.

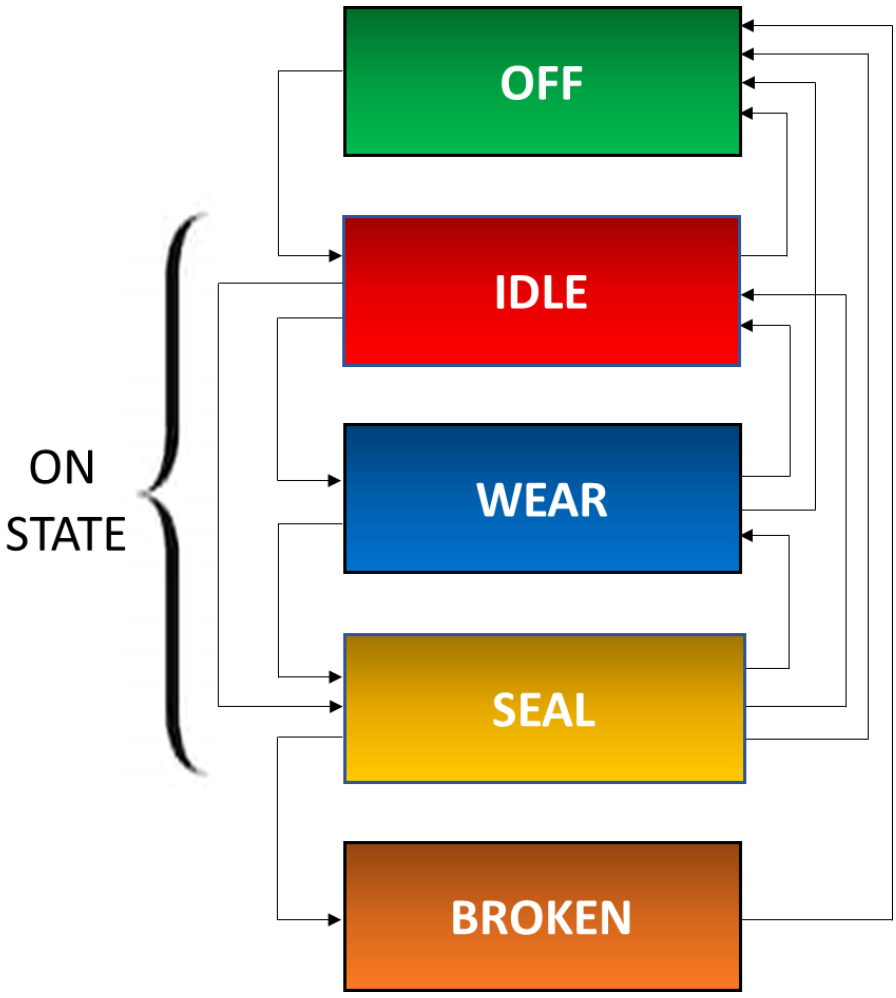

Figura 4-2: Flow Chart of the five main states for the cylinders.

4.2.1 OFF STATE (AND ON STATE)

OFF: when a cylinder is in this state, it is disabled and no operation can be carried out on it, except for enabling the cylinder by pressing the SELECT button, entering the IDLE state. Thanks to the same button, which acts as an ON/OFF switch, it is possible to turn off the cylinder at any time independently of the particular state of the actuator. This happens also in the case of the BROKEN state, even though in that situation it is strongly recommended to substitute the broken parts of the cylinder before trying to make it work again, or change the cylinder itself. During the OFF state all the LEDs are turned off.

ON: this state is actually composed by three main states, namely IDLE, WEAR and SEAL. When the cylinder is on, the SELECT LED (green LED) is turned on (otherwise in all the other states it is turned off).

4.2.2 IDLE STATE

IDLE: this is the initial state in which the cylinder is set by default during the turn on of the test bench, in the "*setup()*" portion of the Arduino code. In this state the cylinder is just inactive, waiting for the user to give a particular order. If we do not take into account the ON/OFF transition, it is possible to exit from this state only by pressing the WEAR button (entering the WEAR state) or the SEAL button (entering the SEAL state). These two passages can happen in the opposite direction (from WEAR to IDLE or from SEAL to IDLE) by pressing twice the STOP button (the first time the button is pressed the cylinder enters into a PAUSE state that will be examined later on). The LED that informs the user that the cylinder is in the IDLE state is the STOP LED (red).

4.2.3 WEAR STATE

WEAR: during this state, a wear test is performed on the cylinder. The piston has to complete a certain number of cycles, and that number is set on the Arduino code before the execution of the program and can be easily changed by the programmer when test

bench is not active. The transitions between the WEAR state and the IDLE state, or between the WEAR state and the OFF state have already been considered. It is also possible a passage between the WEAR state and the SEAL state. When the cylinder has completed the expected number of cycles and the end of the wear test arrives, the transition to the SEAL state is automatic; this passage can also take place if the user pushes the SEAL button during the WEAR state.

The opposite passage, from the SEAL state to the WEAR state, is also automatic. When a seal test comes to its end, another wear test starts. If no button is pressed, the cylinder transitions continuously from the WEAR state to the SEAL state and vice versa, like in an endless loop. Moreover, the seal test cannot be stopped by pressing the WEAR button, in an attempt to start the wear test.

The LED responsible for informing that the air cylinder is in the WEAR state is the WEAR LED (the blue one).

4.2.4 SEAL STATE

SEAL: during this state, a seal test is performed on the cylinder, measuring the pressure of the compressed air inside the cylinder chambers. There are two phases is this state, and two different seal tests that take place: the first test is for the front chamber, the second one is for the rear chamber. The reason behind the two tests is the fact that we need to evaluate both the air leakage towards the outside (in the case of the test on the front chamber) and the leakage between the two chambers (in the case of the test on the rear chamber). The first case is related to the wear of the front end seal, while the other case is connected to a worn out piston seal.

During the seal test, the pressure values of the air inside the cylinders are detected by the pressure sensors, read by the Arduino microcontroller at a specific frequency and sent to the Raspberry Pi, that saves them inside some files. If the pressure goes under a certain value (for example 0.5 bars) the seal test is considered “failed” and the cylinder transitions from the SEAL state to the BROKEN state. The passages between the SEAL state and the other states have already been examined.

The LED that is activated during the SEAL state is the SEAL LED (the yellow one).

4.2.5 BROKEN STATE

BROKEN: this is the state in which the cylinder enters when the seal test is not passed. In this state no operation can be carried out on the cylinder and it is preferable to remove the air actuator, that is considered “broken” because of the air losses, and use another one.

In this state, all the LEDs are turned off, except for the BROKEN LED (red LED, a different shade of red with respect to the STOP LED).

5 ARDUINO AND THE ARDUINO CODE

5.1 INTRODUCTION TO ARDUINO AND ARDUINO MEGA 2560

Arduino is an open-source microcontroller development board. In other words, it is a board used for electronic projects, able to interact with objects by controlling them or acquiring values from them. In order to work, Arduino has to be programmed by using the Arduino software IDE, an Integrated Development Environment that can be downloaded for free and can run both online and offline.

Arduino is very widespread thanks to its flexible and easy-to-use software and hardware. Everyone with basic or no technical knowledge can easily start and learn some skills to program and run the board. One of the main reasons why this board is quite popular, which is also one of the main differences between Arduino and most of the previous programmable circuit boards, is the fact that in order to upload a program on Arduino there is no need to use an additional piece of hardware (called programmer). To load new code it is sufficient to connect a simple USB cable between Arduino and the computer on which the IDE is running. Moreover, learning to program on the IDE is not a hard task because the software uses a simplified version of one of the most common programming languages, C++.

Thanks to the popularity of Arduino, it is very easy to find all over the Internet, besides the official site, a huge amount of guides, blogs and forums created by common users that can turn out to be very helpful to learn the basics and also solve particular problems. This makes the choice of using Arduino even more reasonable.

There are many types of Arduino boards. The most common version of Arduino is the Arduino Uno, the one we used in an initial experimental phase to get acquainted with the device and its functionalities. However, in order to have a sufficient number of I/O pins for our project, in the end we decided to use instead an Arduino Mega 2560, the second most widespread version of Arduino.

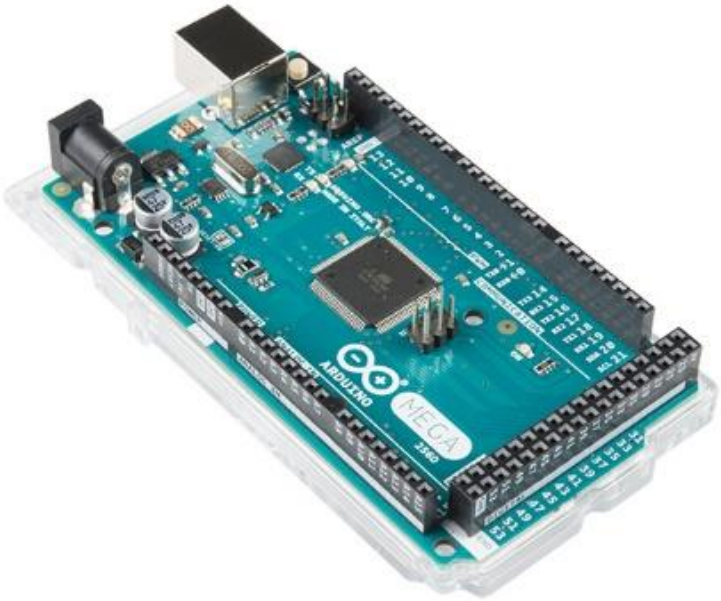

Figura 5-1: Arduino Mega 2560 board

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 chip. This board features 54 digital I/O pins (15 of them can be used for PWM) and 16 analog inputs, almost four time with respect to the pins provided by Arduino Uno. Moreover, it has a crystal oscillator of 16 MHz frequency (CPU frequency), 256 KB of flash memory, 8 KB of SRAM and 4 KB of EEPROM. The board comes with a USB cable port that is used to connect the Arduino to the computer when a transfer of code is needed, but can also be employed to power the board. In our case, however, the power supply for Arduino is granted by the resistive trimmer that provides as output 5 Volts, the operating voltage of the microcontroller. Arduino Mega also comes with two voltage regulators, 5V and 3.3V, that were both required in our work.

As for the pins, we can mention some of the pins featured on the Arduino Mega, in particular the ones required in our project:

- A 3.3 V pin, three 5V pins and five ground pins. Many of these pins were used in our work. We are pointing out this fact since the presence of this type of pins is heavily reduced on the Arduino Uno, so this is an advantage of the Arduino Mega.

- The two pins for I2C communication, used for the PCF8574N integrated circuits as mentioned before.

- Digital pins, used for the valves, the LEDs and the PCFs.

- Analog pins, used for the Reed switches and the pressure sensors, that come with 10-bit resolution and can measure from ground to 5V.

In order to explain which pins were used and why, in the following section we will go into more detail with the Arduino code.

5.2 MAIN STRUCTURE OF THE ARDUINO CODE

The name that Arduino uses for a program is a sketch. A sketch is composed by different parts, but the fundamental characteristic of a code written in Arduino IDE is the presence of two main parts, two main functions that are necessary to the execution of the program. These two functions are called *setup()* and *loop()*.

The *setup()* function is invoked at the beginning of a sketch. It is used to initialize pin modes, variables, start using libraries, serial communication etc. The code inside the *setup()* function runs only once, every time the Arduino board is powered up or reset.

The *loop()* function, instead, is a portion of code meant to be executed repeatedly. After the *setup()* function runs once, the sketch enters the *loop()* function, that immediately follows the *setup()* block. Each line inside the *loop()* block is executed one after another, until the last "}" of the *loop()* block is reached, then the execution comes back to first line of the *loop()* function.

The Arduino sketch we developed also shows other elements.

Before the *setup()* function our code contains the definition and initialization of some variables, some preprocessor directives beginning with the character # (such as #define), inclusion of libraries, definition of new types using the keyword "typedef".

After the *loop()* function we have the definition of some functions and the presence of some ISR, Interrupt Service Routines.

All these elements will be considered more in detail later on.

5.3 CHARACTERISTICS OF THE CODE

Before starting to write the code, we had in mind some of the characteristics that each code should have in order to be considered a good code. We wanted our code to be accessible to others so that it could be easily read, understood, adapted and eventually expanded in the future by other users.

We tried to keep the code clear and concise, efficient by avoiding many useless parts, well-structured by regrouping lines of code into functions and giving each part of the code its own purpose. Of course our code had to be correct, fulfilling the requirements specified for the realization of the test bench and its functionalities, avoiding the occurrence of software errors and unpredicted situations.

In order to have a reusable and extendable code we focused on two characteristics: maintainability and scalability.

A maintainable code is a code that other people can read without any trouble, and this is achieved by using a combination of simple, short, useful and consistent names with specific logic. The code is flexible so that changes can be easily realized in order to fulfill possible new requirements that could come up in the future. For this purpose, at the beginning of the code we have many preprocessor directives by means of the command #define. This directive allows us to give a meaningful name to a constant value before the compilation of the program; during the compilation the words that refer to these constants will be replaced by the compiler with the defined value. Thanks to the use of #define and global variables it is possible to change the value of a parameter only in one line at the beginning of the code, knowing that this will affect the rest of the sketch.

A scalable code, instead, is a code that always works as it is supposed to, even if it works at a higher "scale" of elements. For example if a program does something on a small database, the program can be considered scalable if it is able to work well on a similar larger set. Considering our own particular situation, we are referring to the fact that even though in our test bench currently we have only two cylinders, we want the code to be easily adaptable for the case in which there are more cylinders to operate. This goal is accomplished by using a series of expedients.

5.4 THE *CILINDRO* STRUCTURE

Inside our sketch, the cylinders are considered as equivalent elements defined by similar characteristics and fields. For each cylinder we have a series of heterogeneous data that can vary in time; in the C language, the type of variable that can fit very well this situation is the struct type, because a structure is a cluster of completely unrelated variables that refer to a bigger object, corresponding in our case to a cylinder. By using the primitive typedef we define as struct type a new type, "*cilindro*", knowing that we will have more elements of this particular type. For this reason, we define an array of these elements called "*cilindri*" whose size is indicated by the parameter "N_CIL". In our case, we just specify that N_CIL, that is the total number of cylinders in our test bench, is equal to two by means of the following line:

```
#define N_CIL 2
```

The code related to the type "*cilindro*" and the declaration of the array "*cilindri*" is the following one:

```
typedef struct{
  int state;

  bool TENUTA_ANT;

  long int N;
  long int N_tot;
  long int T0;
  long int T0_blink;
  long int T0_pretenuta;
  long int T0_valveoff;

```

```
13.   bool fineCorsa[2];
14.   int valvePin[2];
15.   int reed[2];
16.   int pressSens;
17.   bool valvePin_prestop[2];
18.
19.
20.   int STOP_LED; //rosso
21.   int SEAL_LED; //giallo
22.   int WEAR_LED; //blu
23.   int SELECT_LED; //verde
24.   int BROKEN_LED; //altro rosso
25.   bool tenutaOK[2];
26.
27.
28.   } cilindro;
29.
30. cilindro cilindri[N_CIL];
```

In this portion, it is possible to recognize a lot of different fields that will all be examined one by one. The advantage of having a similar structure and array is the fact that in the code we can operate some changes on a specific field of all the cylinders at the same time by using for cycles, not having to write the same instruction for each cylinder.

For example, being *N_tot* the total number of strokes (which is the total number of cycles divided by 2) completed by the cylinder since the launch of the sketch, inside the *setup()* block we initialize this variable to the zero value for each cylinder with this piece of code:

```
1.  for(int n_cil=0;n_cil<N_CIL;n_cil++){
2.      cilindri[n_cil].N_tot=0;
3.    }
```

It is also worthy of mention the fact that, in order to make the configuration of the Arduino pins easier in the *setup()* block and to exploit the possibility of using the for cycles, during the project phase we decided to allocate similar resources (for example all the LEDs, all the Reed Sensors, etc.) to pins that were consecutive on the board.

For example, the two pressure sensors were connected to the consecutive pins 54 and 55, corresponding to Analog Pin 0 and Analog Pin 1. The variable related to the pressure sensors inside the *cilindro* structure is "*pressSens*". The code to configure pins 54 and 55 is the following:

```
1.  for(int n_cil=0; n_cil<N_CIL; n_cil++){
2.
3.     cilindri[n_cil].pressSens=n_cil+54;
4.     pinMode(cilindri[n_cil].pressSens, INPUT);
```

```
5.
6.   }
```

In this fragment, it is possible to see the function *pinMode*, fundamental in Arduino to define the pins and declare if they are used as inputs or outputs.

5.5 INTERRUPTS

There are two possible techniques for any microcontroller to recognize the presence of a new input: they are named “polling” and “interrupt”. When the polling method is used, the microcontroller has to check periodically the status of the voltage on the input pins, in order to evaluate the following action to perform. Even if this solution could have some advantages in certain situations, in our case we wanted to avoid high latency (the delay between the time when an event occurs and the time when the microcontroller performs an action due to that event) and an excessive utilization of the CPU time to continuously check the pins. We decided to adopt interrupts.

An interrupt is an asynchronous signal generated by the variation of the status of a pin that stops the execution of the *loop()* function so that an event can be handled. The regular flux of instructions goes under an interruption and a routine called Interrupt Service Routine (ISR) is invoked. When the routine ends, the running program continues normally with the next instruction.

One of the drawbacks of the use of interrupts is the fact that during the execution of an Interrupt Service Routine other interrupts are disabled and some fundamental functions such as *delay()* and *millis()* do not work. These functions are important in software debouncing of the buttons, so the use of interrupts is also one of the reason why we adopted hardware debouncing.

There are different types of interrupt implemented on an Arduino Board. The basic interrupts in Arduino, exposed directly on the Arduino pins, are the external interrupts, that we chose not to use. The interrupts we employed instead are timer interrupts (that will be discussed later on) and pin change interrupts.

5.5.1 PIN CHANGE INTERRUPTS

When pin change interrupt is enabled on a pin, it means that whenever the status on that particular pin goes through a modification an interrupt will be sent. Only some ports on the Atmega2560 (the microprocessor in the Arduino Mega we used) support pin-change interrupts, and they are Port B, Port E (only bit 0), Port J (bits from 0 to 6) and Port K. We decided to use only Port B and Port K, and only some pins on them. To enable pin change interrupts the following lines of code were written before the *setup():*

```
1.  #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
2.  #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
```

And inside the *setup()* :

```
1.  //PREPARE PIN CHANGE INTERRUPT INTERRUPTS
2.
3.    //PORTB
4.    sbi(PCICR, PCIE0);
5.    sbi(PCMSK0, PCINT0); //DIGITAL PIN 53
6.    sbi(PCMSK0, PCINT1); //DIGITAL PIN 52
7.
8.    //PORTK
9.    sbi(PCICR, PCIE2);
10.   sbi(PCMSK2, PCINT18); //DIGITAL PIN 62
11.   sbi(PCMSK2, PCINT19); //DIGITAL PIN 63
12.   sbi(PCMSK2, PCINT16); //DIGITAL PIN 64
13.   sbi(PCMSK2, PCINT17); //DIGITAL PIN 65
```

From this portion of code, remembering what we mentioned in the previous chapters, we can see that we connected the PCFs to Port B and the Reed sensors to Port K.

In this fragment, we have some registers where we are setting some bits by means of *sbi*.

It is important to know that each pin on the same port shares the same interrupt signal, this means that in order to detect which particular pin actually toggled and provoked the interrupt, the register of the port that contains that pin has to be read. In particular, Port B is associated with the Pin Change Interrupt 0 (PCI 0), while port K is associated with Pin Change Interrupt 2 (PCI 2). To enable them, bits PCIE0 and PCIE2 have to be set in the PCICR register. Then for both PCI 0 and PCI 2 there is a mask, a register called

respectively PCMSK0 and PCMSK2 in which we set the particular bits related to the pins for which we want to enable pin change interrupts.

Each of these pin change interrupts has its own “vector” to call the related Interrupt Service Routine. We have ISR(PCINT0_vect) for PCI 0 and ISR(PCINT2_vect) for PCI 2.

We had to define some variables that we used in the two ISR:

- PortBStatus/PortKStatus: they have the dimension of a byte; in these variables we store the bits of PINB/PINK, the registers that contain the momentary status of the pins of Port B/Port K, in order to make some changes on those values.
- PortBUpdated/PortKUpdated: Boolean variables; they act as flags, becoming true when an interrupt arises and returning to a false value inside the *loop()* function, where specific operations are performed to deal with the interrupts. These variables are initialized to “false” in the *setup()* block.

We will consider the two different ISR we have for these two PCI by examining first how the PCFs were managed inside our sketch, and then we will consider the management of the Reed sensors.

5.5.2 MANAGEMENT OF THE PCF8574N INTEGRATED CIRCUITS

The code related to the ISR associated to Port B and the PCFs is the following one:

```
1.  ISR(PCINT0_vect)
2.  {
3.    PortBStatus=PINB;
4.    PortBStatus=~PortBStatus;
5.
6.    PortBStatus&=0x3;
7.
8.    if(PortBStatus==1 ||PortBStatus==2){
9.
10.     PortBUpdated=true;
11.   }
12. }
```

(N.B.: in this case, as in many others, we will not take into consideration the *Serial.print* lines that print data to the serial monitor to give us information about what is happening during the execution of the sketch)

The content of the register PINB is copied inside PortBStatus. The instruction "PortBStatus&=0x3" corresponds to "PortBStatus = PortBStatus & 0x3". Considering that having 0x3 is like having the sequence of bits "00000011", we are using a mask that makes meaningless each bit of the port except for the first two bits, the only two bits that we care about (PB0 and PB1 - 53 and 52 - the two pins we used to connect the PCFs). The two bits are normally high and become low when an interrupt is generated inside the PCF. By using the instruction "PortBStatus = ~ PortBStatus" (the tilde character corresponds to the bitwise not) we make it so that when an interrupt is sent by PCF1 PortBstatus is equal to 1 ("00000001"), if it is sent by PCF2 instead it is 2 ("00000010"). By using the control "if(PortBStatus==1 || PortBStatus ==2)" we simply discharge the interrupts generated inside the pins of Arduino that are due to the return to the original voltage of the interrupt pins of the PCFs. If one of the two occurrences inside the if control happens, PortBUpdated is set to true and the ISR ends. Then PortBUpdated will be examined in the *loop()* block.

In order to make the communication possible between Arduino and the two PCFs, inside our code we had to include a library by means of the instruction "#include <Wire.h>", call the *Wire.begin()* function inside the *setup()* block to start the communication, then we also had to write two functions, that make use of the functions contained in the Wire.h library:

```
1.  void expanderWrite(byte _data, byte address ) {
2.    Wire.beginTransmission(address); //es address = 0x42>>1
3.    Wire.write(_data);
4.    Wire.endTransmission();
5.  }
6.
7.  byte expanderRead(byte address) {
8.    byte _data;
9.    Wire.requestFrom(address, 1);
10.   if(Wire.available()) {
11.     _data = Wire.read();
12.   }
13.   return _data;
14. }
```

The “expanderWrite” function allows to write on the pins of a PCF by specifying as parameters the address of the PCF and the sequence of 8 bits that we want to write. We used this function only during an initial test phase but we are leaving it in the code because it could be useful for future developments.

The “expanderRead” function has the purpose of allowing the reading of the pins of a PCF. The only parameter featured is the address of the PCF, and the return value is a byte variable containing the status of all the eight pins of the PCF.

Some considerations have to be made about the way we write the address of the PCFs inside our sketch. We know from a previous chapter that we chose a “000” address for PCF1 and “001” address for PCF2. Reading the datasheet of the PCF [4], we discover that the complete addresses are represented by 8 bits, some of which are already fixed, as shown in Table 5-1. We find out that the addresses we chose, that have to be written in HEX, correspond to the hexadecimal numbers 0x40 and 0x42. However, the Wire-library shifts internally the bits one to the left and sets bit 0 to “0” [8]. If we wrote in our code 0x42, it would become 0x82 and the PCF would not respond to the messages. So when writing the addresses we have to write 0x40>>1 and 0x42>>1, shifting the bits of the address one to the right.

BYTE	**BIT**							
	7 (MSB)	**6**	**5**	**4**	**3**	**2**	**1**	**0 (LSB)**
I²C slave address	0	1	0	0	A2	A1	A0	R/W
I/O data bus	P7	P6	P5	P4	P3	P2	P1	P0

Table 5-1: I2C interface definition

Therefore, these addresses are passed in our code to the “expanderRead” function, which is called inside our code inside another function named “GetSwitchPress”. This last function returns an integer value and receives as a parameter a character. This character is used to refer to a port rather than the other one. The character used for port B is ‘”B”. The code in which “GetSwitchPress” is called is the following one:

```
1.  if(PortBUpdated){
```

```
2.
3.        PortBUpdated=false;
4.        butt=GetSwitchPress('B');
5.      }
```

This portion is at the beginning of the *loop()* section. After the operations done in the ISR in order to take into account only the meaningful pin change interrupts, PortBUpdated is set to true. At the beginning of the *loop()* block we check if PortBStatus is true: if it is so, we set it to false and we invoke the GetSwitchPress function for the Port B. The return value is stored into a variable named “butt”.

“butt” is an integer-type global variable defined before the setup and initialized to the value -1, the default value for this variable. Each time a button is pressed the value of this variable changes thanks to the GetSwitchPress function, then some instructions are executed inside the *loop()* block, and then at the end of the *loop()* the value of “butt” is set again to -1. This last operation is done in order to avoid that the operations done consequently to the pressing of a button are executed more than once by mistake when the button, instead, has been pressed only one time.

We will now take into consideration the part of the GetSwitchPress function related to Port B and the PCFs. The code is the following one:

```
1.  int GetSwitchPress(char Port){
2.    if(Port=='B'){
3.     if(PortBStatus==1){ //53
4.
5.          byte data = expanderRead(0x40>>1);
6.          data = ~data;
7.          byte SwitchState = data;
8.          switch(SwitchState){
9.
10.           case 1:
11.           return 12000;
12.           case 2:
13.           return 220;
14.           case 4:
15.           return 11000;
16.           case 8:
17.           return 210;
18.           case 16:
19.           return 310;
20.           case 32:
21.           return 110;
22.           case 64:
23.           return 320;
24.           case 128:
25.           return 120;
26.         default:
27.           return -1;
```

```
28.         }
29.       }
30.       if(PortBStatus==2){ //52
31.
32.         byte data = expanderRead(0x42>>1);
33.         data = ~data;
34.         byte SwitchState = data;
35.         switch(SwitchState){
36.
37.           case 1:
38.           return 10000;
39.           case 2:
40.           return 200;
41.           case 64:
42.           return 300;
43.           case 128:
44.           return 100;
45.
46.         default:
47.           return -1;
48.         }
49.
50.       }
51.     }
52. }
```

The first “if” condition is verified because Port is equal to “B” when the function is called in the *loop()* block, as we have seen before. Then we have two different conditions depending from PortBStatus. If PortBStatus is equal to 1, as we have mentioned previously, the PCF that sent the interrupt is the PCF1. At this point, we know which pin of Port B has sent the pin change interrupt (a meaningful one) and the PCF that has sent the interrupt, but this is not enough: we have to read the status of the pins of the PCF in order to find out which button has been pressed. As we know, when a button is pressed the voltage level of the pin goes from high to low, so we read the content of the pins and store it inside the variable “data”. The byte stored in this variable will be composed by a series of “1” bits except for the bit corresponding to the button pressed. We want to read an opposite situation, having “1” for the relevant bit and “0” for the others, so we apply the tilde operation on the variable “data”. Then this new “data” will be discussed with the new name of SwitchState inside a switch-case. If the first bit which is bit 0 is 1 we have “case 1” (2^0=1), then if bit 1 is “1” we have “case 2” (2^1=2), then so on and so forth, until the bit 7 for which we have “case 128” (2^7=128). For PCF2 the situation is completely equivalent, we will just take into account the four pins related to the four buttons we attached to the second PCF.

The default return value of the GetSwitchPress function is -1. The other return values are other numbers chosen randomly that will be examined inside a function called update_butt(), whose content will be explained later on.

5.5.3 MANAGEMENT OF THE REED SENSORS

As for the Reed sensors, we have defined inside the *cilindro* structure an array of two components, where we store the values of the pins to which we connected the Reed switches. For each cylinder we have the same array "`int reed[2];`". The pins for the Reed Sensors are 62 and 63 for the first cylinder, 64 and 65 for the second cylinder. 62 and 64 refer to the same side of the cylinder (Reed sensors turning on when the piston is fully retracted), while 63 and 65 correspond to the other side. Inside the *setup()* block we define the pins as input by means of the following fragment:

```
1.   for(int n_cil=0; n_cil<N_CIL; n_cil++){
2.
3.      for(i=0;i<2;i++){
4.        cilindri[n_cil].reed[i]=n_cil*2+i+62;
5.        pinMode(cilindri[n_cil].reed[i], INPUT);
6.        cilindri[n_cil].fineCorsa[i]=digitalRead(cilindri[n_cil].reed[i]);
7.      }
8.  }
```

Inside this portion, we also initialize the content of some "fineCorsa" variables using the current status of the Reed sensors. The meaning of these variables will be explained later in this paragraph.

The ISR related to Port K and the Reed sensors is the following one:

```
1.  ISR(PCINT2_vect)
2.  {
3.    PortKStatus=PINK;
4.    PortKStatus&=0x0F;
5.
6.    if(PortKStatus>0){
7.
8.      PortKUpdated=true;
9.
10.   }
11. }
```

Similarly to what we have seen for the PCFs, we take the content of PINK, the register that contains the momentary status of Port K, and we store it inside PortKStatus. The pins of Port K show a high voltage level when the Reed switches are active, so there is no need to reverse the bits of PortKStatus with a tilde operator. We use a mask in order to take into consideration only the content of the first four bits; for this purpose we use the hexadecimal number 0x0F (00001111). At this point, the flag PortKUpdated is set to 1 whenever PortKStatus is bigger than 0, discharging only the case in which no Reed sensor is active (pistons in the midst of the stroke), not considering all the smaller cases. This is due to the fact that we can have more than one Reed sensor active at the same time (of course in that case they are Reed sensors applied to different cylinders) so we would have to examine a lot of smaller cases corresponding to situations when one or more than one reed is active. This kind of approach becomes unsustainable if we have to deal with a higher number of cylinders and Reed sensors, so we rejected it.

We deal with the updates of Port K inside the *loop()* function. The code is the following one:

```
1.  else if(PortKUpdated){
2.
3.      byte temp=PortKStatus;
4.      PortKStatus=(PrevPortKStatus^PortKStatus)&PortKStatus;
5.      PrevPortKStatus=temp;
6.
7.      PortKUpdated=false;
8.
9.       byte checkbit=1;
10.      for(int n_cil=0;n_cil<N_CIL; n_cil++){
11.       for(int i=0; i<2;i++){
12.
13.           if((PortKStatus&checkbit)>0){
14.             cilindri[n_cil].fineCorsa[i]=true;
15.
16.           }
17.           else{
18.             cilindri[n_cil].fineCorsa[i]=false;
19.
20.           }
21.           checkbit=checkbit*2;
22.     }
23. }
```

The initial “else if” is connected to the “if” condition that discusses about the value of PortBUptated. If PortKUptated is true, it has to be set to false. The first three lines inside the “else if” block makes it so that we analyze a modified PortKStatus in order to update the content of some variables called “fineCorsa”. These boolean variables act as flags:

they have to become true when the piston reaches the end of stroke, then the information is used, then the variables can become false. There are as many “fineCorsa” variables as the Reed sensors. They are defined inside the *cilindro* structure using an array of two elements, “`bool fineCorsa[2];`”. The first element, “fineCorsa[0]”, refers to the Reed that turns on when the piston rod is fully retracted, while the second element, “fineCorsa[1]”, refers to the Reed that activates when the piston rod is fully extended.

We have a byte variable called PrevPortKStatus defined before the *setup()* and initialized inside the *setup()* with the first four bits of the PINK register by means of the instruction “PrevPortKStatus = PINK&0x0f”. Inside the variable PrevPortKStatus we store the content of PINK prior to the incoming of the interrupt. We compare it by means of the EXOR bitwise operator to the current content of PortKStatus, which is stored in a support variable called “temp” and then becomes the new PrevPortKStatus. The resulting bits of the EXOR are 1 if a change in that bit has occurred, 0 if the bit has not changed. Then, by means of the AND bitwise operator, at the end we have a PortKStatus that only takes into account the changes from 0 to 1. The three lines that act on PortKStatus have been added because we want that, when the “fineCorsa” variable becomes true after the activation of a Reed sensor, the information provided by “fineCorsa” is exploited during that particular loop cycle and not used more than once. Even if the Reed sensor is still on, we are discharging the 1-1 bit transition because we only care about the first activation of the Reed switch, we do not want the “fineCorsa” variable to become true again if actually there is no new information. “fineCorsa” becomes zero, except for the case when there is a 0-1 bit transition.

We use two for cycles, one for the different cylinders and one for the different Reed sensors of the same cylinder. We create a support byte variable named “checkbit” to check bit by bit the modified PortKStatus, from the first bit to the last that we need to consider (in our case, four bits). This is done by updating the content of checkbit at each cycle, going from 1 (0001) to 2 (0010), then 4 (0100), then 8 (1000). At each cycle, we also modify the content of the “fineCorsa” variables, according to the logic we have just explained.

5.5.4 TIMER INTERRUPT AND MANAGEMENT OF THE PRESSURE SENSORS

As we have already mentioned, the two pressure sensors were connected to the consecutive pins 54 and 55, corresponding to Analog Pin 0 and Analog Pin 1. The variable related to the pressure sensors inside the *cilindro* structure is "*pressSens*". The code to configure pins 54 and 55 is the following:

```
1.  for(int n_cil=0; n_cil<N_CIL; n_cil++){
2.
3.     cilindri[n_cil].pressSens=n_cil+54;
4.     pinMode(cilindri[n_cil].pressSens, INPUT);
5.
6.   }
```

During the seal tests, we need to read the analog values coming for the pressure sensors using a certain frequency. In order to specify this frequency we use a particular tool featured in Arduino and called "Timer Interrupt".

As we know, inside the Arduino microcontroller there is a quartz crystal that acts as an oscillator, generating pulses with a specific speed. The speed of the oscillator is 16 MHz, so this means that each pulse comes with specific time, 1/16M seconds. Therefore, there is a constant duration between two consecutive pulses. If we count the number of pulses with a counter, we can calculate real time. The hardware part responsible of the counting task in Arduino is called "Timer", and it is a counter that increases with a particular rate, so that it is possible to detect how much time passes knowing the value of the counter.

The timer in an MCU is a register that consists of a specific number of bits. In Arduino Mega we have 6 timers named Timer0, Timer1 etc. till Timer5. They are all 16 bits timers except for the 8 bits timers Timer2 and Timer0. These timers differ in the timer resolution: 8 bits means 256 values for the counter, while 16 bits timers give a higher resolution, with 65536 values.

Timer0 is used by the microcontroller for the timer functions delay() and millis(). Since we needed to use these standard functions in our code, the best choice was to avoid to use Timer0. We employed instead the 16 bits timer Timer1.

Timers generate interrupts at a specified frequency, which is always lower than the oscillator frequency. In order to change the behavior of the timer there is a series of timer registers that have to be configured. These are the most important:

- TCCRx, the Timer/Counter Control Register, where the configuration of the prescaler happens
- TCNTx, the Timer/Counter Register, where the actual timer value is stored
- OCRx, the Output Compare Register
- ICRx, the Input Capture Register
- TIMSKx, the Timer/Counter Interrupt Mask Register, used to enable or disable timer interrupts
- TIFRx, the Timer/Counter Interrupt Flag Register, that indicates that a timer interrupt is pending

In this list, the "x" notation refers to the number of the timer (we work with Timer1, so x is 1, TCNTx in this case is the register TCNT1).

As we have mentioned in this list, TCCRx is used to set the prescaler, a number by which the 16 MHz frequency of the oscillator is divided in order to obtain a timer at a lower frequency. The prescaler can be described as the number of counts the timer has to make in order to increase its value by 1. Therefore, if the prescaler is set to 8 it means that every 8 clocks of the timer the timer counter value will increase by 1. The larger is the prescaler value, the lower is the frequency of the timer, and the longer is the time that the timer counter can reach before overflowing.

The timer can work in different modes: for example Pulse Width Modulation (PWM) mode or CTC mode, Clear Timer on Compare match, that means that when the timer counter reaches the compare match register, the timer will be cleared. We are using CTC mode.

A timer can generate different types of interrupts. To enable or disable them, the bits of the Timer Interrupt Mask registers (TIMSKx) have to be respectively set or cleared. At

the occurrence of an interrupt, a flag is set inside the Timer Interrupt Flag Registers (TIFRx) and will be automatically cleared when the ISR starts.

There are three different ways of using a timer: if we are checking when the timer has reached its limit value and has overflowed, we will use timer overflow interrupts. There are also timer input capture interrupts, but the type of interrupts we chose to adopt is the third one: output compare match interrupts. By using this technique, we compare the timer count to a specific value, so that every time the count of the timer is equal to that value, an interrupt is sent.

For each timer, it is possible to compare the timer counter value to two different values, stored in the OCRx registers. In our case, we chose to use only OCR1A register and the related ISR named “ISR(TIMER1_COMPA_vect)”.

In our sketch, we created a function named “timerInterruptSetup()” to configure the various registers and initialize the timer, following the instructions of the Arduino Mega data sheet. The code is the following one:

```
1.  void timerInterruptSetup(){
2.      cli();  //disable the global interrupt
3.      //Timer/Counter 1
4.      TCCR1A = 0x00;
5.      //TCCR1B = (_BV(WGM12)) | (_BV(CS11)) | (_BV(CS10));  //CTC mode, clk/64
6.      TCCR1B = (_BV(WGM12)) | (_BV(CS12)); //CTC mode, Set prescaler to 256
7.      OCR1A = OUTPUT_COMPARE; //set timer frequency
8.      TCNT1 = 0x00; //initialize the counter
9.      TIMSK1 = _BV(OCIE1A);  //Output Compare Match Interrupt Enable
10.     sei(); //enable global interrupt
11. }
```

This function is invoked during the *setup()* block. In order to change the values of some registers we are using the _BV() macro, that converts a bit number into a byte value. For example it is true to say that _BV(2) corresponds to 1 << 2 and also to 0x04 (00000100).

To use the timer counter in CTC mode we have to set the WGM12 bit inside the TCCR1B register. To have 256 as a prescaler, inside the same register we have to set the CS12 bit, while we set CS10 and CS11 in order to have 64 as a prescaler. We chose 256.

We initialize the counter by giving a zero value to the TCNT1 register.

In order to use the value stored in OCR1A as value of comparison, we set the OCIE1A bit inside the TIMSK1 register. The value inside the OCR1A register is named "OUTPUT_COMPARE" and we define it before the *setup()* block by means of the line:

```
#define OUTPUT_COMPARE 0x3D09
```

This is the parameter that we have to change to modify the frequency of the timer. In order to choose the hexadecimal values corresponding to OUTPUT_COMPARE, we make some calculations. We know that the CPU frequency is 16 MHz. If we divide it by the chosen prescaler, in our case 256 we obtain 16000000/256 = 62500. If we use as comparison threshold 62500 it will be reached one time in one second, so the timer will sent one output compare match interrupt per second and the frequency of the timer will be 1 Hz. If we set OUTPUT_COMPARE to 31250 (62500/2), the threshold will be reached twice in a second, there will be two interrupts per second, so the frequency will be 2 Hz. To have a 4 Hz frequency we need to choose for OUTPUT_COMPARE the number 62500/4 = 15625, that corresponds to the hexadecimal value 0x3D09. The lower the value in OCR1A, the higher the frequency, the higher the number of samples we will have for the pressure of the chambers of the cylinders. For our seal test, we know that we only need a general trend of the pressure, so we will try to keep the value of OUTPUT_COMPARE not so low.

In order to complete the discussion about the pressure sensors and timer interrupt we need to analyze what happens inside the timer interrupt ISR. The code is the following one:

```
1.  ISR(TIMER1_COMPA_vect){
2.
3.  for(int n_cil=0; n_cil<N_CIL; n_cil++){
4.       if(cilindri[n_cil].state==TENUTA){
5.
6.          float p =(float)(analogRead(cilindri[n_cil].pressSens)-
    ANALOG_TRANSLATION)*ANALOG_TO_BAR;
7.
8.          String X = String(n_cil);
9.          X= X + X;
10.
11.         X = X + "," + String(((float)(millis()-
    cilindri[n_cil].T0)/1000)) + "," + String(p);
12.         Serial.println(X);
13.
14.
15.        }
```

```
16.    }
17. }
```

The first operation inside the interrupt service routine is a check that aims to find out if the cylinder is in a particular state named “TENUTA”, because this is the state in which the seal test is performed. We only want to read the pressure values during the seal test.

Then we need to make a conversion of the values we read from the analog pins of the sensors in order to get the real pressure values. From the datasheet of the pressure transducers we know that the operating pressure range is 0 ÷ 10 bars, while the output voltage ranges from 1 V to 5 V, so 1 Volt corresponds to 0 bars while 5 V corresponds to 10 bars.

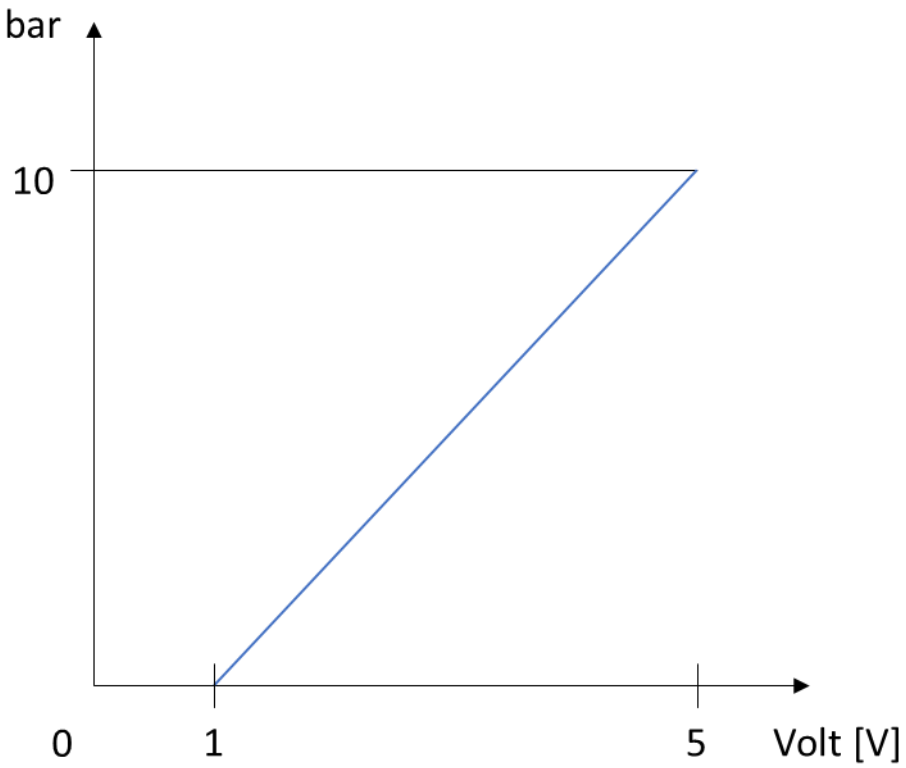

Figure 5-2: Pressure transducer curve

We imagine that the curve relating the pressure and the voltage is approximately a straight line, of which we want to calculate the angular coefficient. This fact is not completely exact, but the approximation becomes more faithful to reality at both ends

of the curve, so around 0 bars and 10 bars. Knowing that 10 bars is the pressure at which our cylinders are supposed to work most of the time, we will consider 10 bars as the working point, and the difference between the real curve and the straight line will be neglected.

Arduino board contains a multichannel, 10-bit analog to digital converter. This means that it will map input voltages between 0 and the 5V operating voltage into integer values between 0 and 1023 (2^10 = 1024). Using a proportion we know that 1 V corresponds approximately to 204 (1023/5). To simplify the conversion, we translate the line relating pressure and voltage into the origin of the axes, by subtracting 204. It is like reading values from 0 to 819, from 0 V to around 4 Volts.

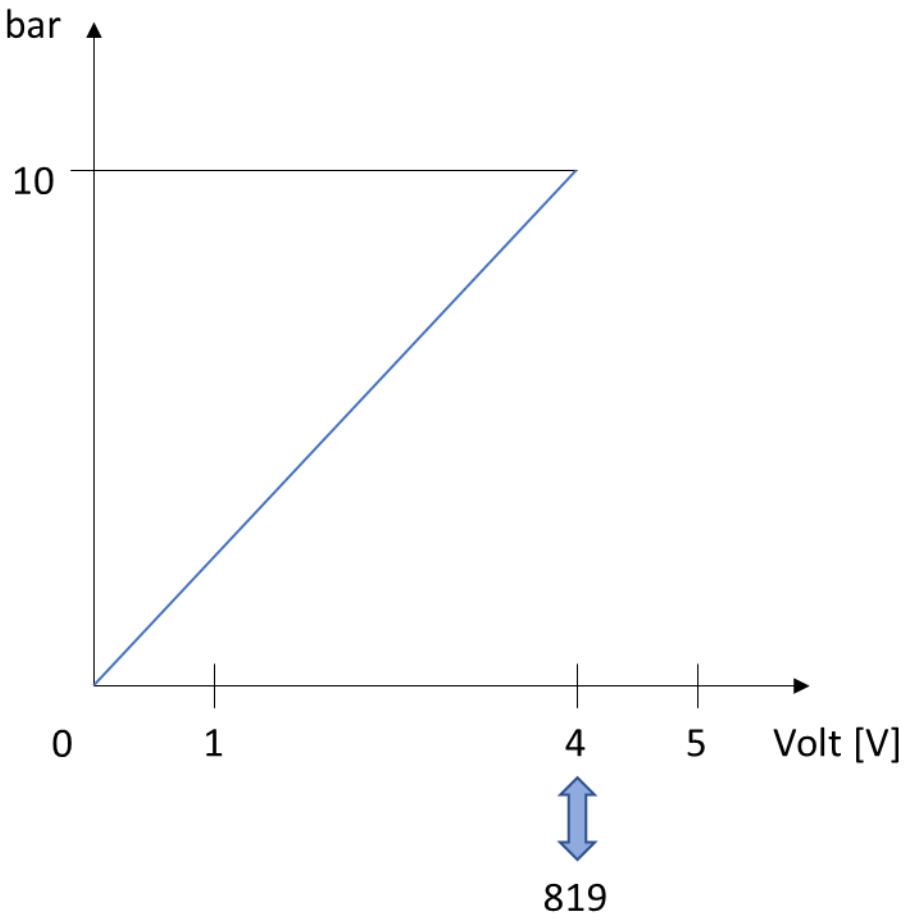

Figure 5-3: Pressure transducer curve after translation

Then, to calculate the angular coefficient we divide 10 (pressure at the working point) by 819 (corresponding analog value) and we obtain 0.0122 as coefficient. We define the values of the translation and the coefficient before the *setup()* function, so we can also modify them in the future if they don't work properly, for example with other pressure sensors. We name these values "ANALOG_TRANSLATION" and "ANALOG_TO_BAR".

```
1.  #define ANALOG_TRANSLATION 204
2.  #define ANALOG_TO_BAR 0.0122
```

Therefore, by subtracting "ANALOG TRANSLATION" to the values we read from the analog pins and then multiplying the result by "ANALOG_TO_BAR", we obtain the pressure values that we store inside a float variable named "p".

The remaining part of the ISR of the timer interrupt aims to prepare the strings containing the pressure values that are sent to the Raspberry Pi, in the form that the strings need to have in order to be saved in the files that Raspberry Pi will create. The strings for the first cylinder have to begin with "00," while the strings for the other cylinder will start with "11,". This is done to help Raspberry recognizing from which cylinder the information is coming, so that the board can save the information in the proper file, related to the proper cylinder.

For the construction of the for cycle, inside the line "`String X = String(n_cil);`" n_cil can be 0 or 1, and it is forced to be cast as a string. By doing "X = X+X", X becomes "00" or "11". Then, the final string that is sent to the Raspberry Pi is prepared. It contains the value of p at the end of the string separated from the rest by means of a comma.

Each pressure value p is related to a particular time instant of the seal test. The time instant considered can range from zero to a final value that coincides with the duration of the seal test, set in the first part of the sketch. The time instants are the second part of the string. The difference between millis() and cilindri[n_cil].T0 is given in milliseconds, so in order to get the time instants in seconds we divide this difference by 1000.

The function millis() is a standard function in Arduino that returns the number of milliseconds passed since the Arduino board started to run the current program. This

number will go back to zero (overflow), after approximately 50 days. *T0* is the field of the *cilindro* structure that stores a particular value of time (in milliseconds), that is the time that has passed between the moment when Arduino began running the sketch and the instant when the current seal test started. T0 is declared as a “long int” variable, and this is because the variables storing instants of time usually require more space. The difference between millis() and T0 goes exactly from zero to the duration of the seal test.

The string obtained in this way is sent to the Raspberry Pi board by means of the command “Serial.println”.

5.6 MANAGEMENT OF THE LEDS

As we already know, we have five different types of LEDs. For each of them, we declare a variable inside the cilindro structure.

```
1.  int STOP_LED; //rosso
2.  int SEAL_LED; //giallo
3.  int WEAR_LED; //blu
4.  int SELECT_LED; //verde
5.  int BROKEN_LED; //altro rosso
```

The LEDs are connected to the digital pins from 2 to 11. In particular, we have:

- pin 2 and pin 3 for the STOP LEDs (pin 2 for the first cylinder, pin 3 for the other)
- pin 4 and pin 5 for the SEAL LEDs
- pin 6 and pin 7 for the WEAR LEDs
- pin 8 and pin 9 for the SELECT LEDs
- pin 10 and pin 11 for the BROKEN LEDs (only in this case pin 11 is for the first cylinder while the previous pin is for the second cylinder).

In order to configure all these pins as outputs, we wrote the following code in the *setup()* block:

```
1.  for(int n_cil=0; n_cil<N_CIL; n_cil++){
2.
```

```
    cilindri[n_cil].STOP_LED=n_cil+2;
    pinMode(cilindri[n_cil].STOP_LED, OUTPUT);
    digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
    cilindri[n_cil].SEAL_LED=n_cil+4;
    pinMode(cilindri[n_cil].SEAL_LED, OUTPUT);
    cilindri[n_cil].WEAR_LED=n_cil+6;
    pinMode(cilindri[n_cil].WEAR_LED, OUTPUT);
    cilindri[n_cil].SELECT_LED=n_cil+8;
    pinMode(cilindri[n_cil].SELECT_LED, OUTPUT);
    digitalWrite(cilindri[n_cil].SELECT_LED, HIGH);
    cilindri[n_cil].BROKEN_LED=11-n_cil;
    pinMode(cilindri[n_cil].BROKEN_LED, OUTPUT);

  }
```

In this fragment, we can see that when the program starts we turn on the STOP LEDs and the SELECT LEDs, while the others are off. This happens because, as we said before, the initial default state is the IDLE state.

During the execution of the program, there are three conditions in which each LED can be: on, off or blinking. These conditions depend on the particular state in which the cylinder is. When the LED is blinking, it switches continuously between the on state and off state at a specific rate. We wanted the possibility to have different rates for different states of the cylinders. To make the LEDs blink we wrote some functions, taking into account only the LEDs that we wanted to provide with this condition. Those LEDs are the SEAL LED, the WEAR LED and the STOP LED. The code for the functions is the following one:

```
void blinkSealLed(int n_cil, int T){
  if(millis()-cilindri[n_cil].T0_blink>T){
    digitalWrite(cilindri[n_cil].SEAL_LED, !digitalRead(cilindri[n_cil].SEAL_LED));
    cilindri[n_cil].T0_blink=millis();
  }
}


void blinkWearLed(int n_cil, int T){
  if(millis()-cilindri[n_cil].T0_blink>T){
    digitalWrite(cilindri[n_cil].WEAR_LED, !digitalRead(cilindri[n_cil].WEAR_LED));
    cilindri[n_cil].T0_blink=millis();
  }
}

void blinkStopLed(int n_cil, int T){
  if(millis()-cilindri[n_cil].T0_blink>T){
    digitalWrite(cilindri[n_cil].STOP_LED, !digitalRead(cilindri[n_cil].STOP_LED));
    cilindri[n_cil].T0_blink=millis();
  }
}
```

Each one of the three functions is perfectly equivalent to the others, so we can consider just one of them. As parameters, we pass:

- *n_cil*, refers to the cylinder to which the LED belongs.
- T, the time the LED spends being on, that is also equal to the time in which the LED is off.

"TO_blink" is one of the fields of the "cilindro" structure. The return value of the millis() function (we already saw what this value is) is assigned periodically to the T0_blink variable. The difference between millis() and T0_blink becomes greater than T when a time T has passed since the last update of T0_blink. After this time T we read the status of the LED pin and we toggle it by means of the "!" operator. If the LED is on it becomes off and vice versa. Then inside the "if" condition we also initialize again the value of T0 so the "if" condition will become true again after time T. The greater the time T that we choose as parameter, the lower the blinking rate.

5.7 STATES OF THE CYLINDERS

As we have already discussed in a previous chapter, there are many states in which a cylinder can be. The variable "state" declared inside the *cilindro* structure is a variable that stores integer numbers, each one of them corresponding to a particular state of the cylinder. The numbers are chosen randomly and just have the purpose of discriminating one state from the other inside the integer variable. These numbers are related to meaningful names by means of some directives:

```
1.  #define USURA 0
2.  #define TENUTA 1
3.  #define IDLE_STATE 3
4.  #define SWITCH 4
5.  #define ZEROING_USURA 5
6.  #define PRE_TENUTA 6
7.  #define ZEROING_TENUTA 7
8.  #define VALVE_OFF 8
9.  #define PAUSA_USURA -1
10. #define PAUSA_TENUTA -2
11. #define ROTTO -3
12. #define OFF -4
```

We will now give more details about each of these states:

- OFF - this state corresponds exactly to the OFF state described in chapter 4. When the cylinder is off, no operation can be performed except for enabling the air actuator, entering IDLE_STATE
- IDLE_STATE - this state corresponds exactly to the IDLE state described in chapter 4. The cylinder is on, waiting for the user to do something. It is the initial default state. Inside the *setup()* block we have:

```
for(int n_cil=0;n_cil<N_CIL;n_cil++){

cilindri[n_cil].state=IDLE_STATE;

}
```

- ROTTO - this state corresponds exactly to the BROKEN state described in chapter 4. The cylinder has not passed the seal test so no operation can be executed on it.
- USURA - it is the state in which the wear test takes place, and the cycles are completed. Along with ZEROING_USURA, it is part of the WEAR state described in chapter 4.
- ZEROING_USURA - it is an initial phase of the wear test useful to have the piston rod fully retracted at the beginning of the wear test. Along with USURA, it is part of the WEAR state described in chapter 4. After this state, the cylinder automatically enters the state of USURA.

The flow chart regarding the WEAR state is the following one:

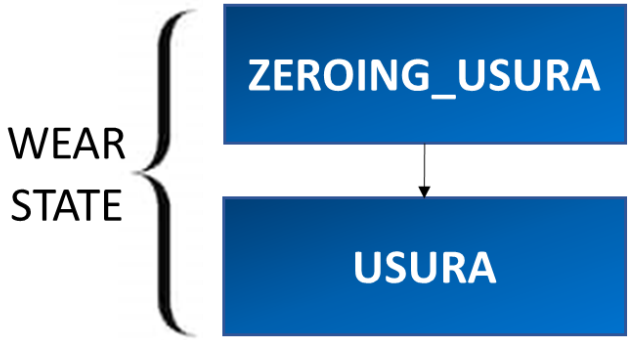

Figure 5-4: States of the WEAR state

We will now take into consideration what actually happens inside the SEAL state described in chapter 4. The flow chart describing the situation is the one in the following page, where the passages are all automatic and not due to the pressing of a button. The states involved are:

- TENUTA - it is the state in which the actual seal test takes place. It is part of the SEAL state described in chapter 4, along with ZEROING_TENUTA, VALVE_OFF, PRE_TENUTA and SWITCH. During the normal execution of the SEAL state, the cylinder passes through the TENUTA state two times; the first time is for the front chamber, while the second time is for the rear chamber. Both of the times the cylinder enters the TENUTA state after being in the PRE_TENUTA state. In the TENUTA state, valves are closed.
- ZEROING_TENUTA - it is an initial phase of the seal test useful to have the piston rod fully retracted at the beginning of the seal test for the front chamber.
- VALVE_OFF - in this state, the valves are closed for a brief time to let the air inside the cylinder stabilize before the seal test. During the normal execution of the SEAL state, the cylinder enters the VALVE_OFF state twice; the first time is after the ending of ZEROING_TENUTA, the second time is after the ending of the SWITCH state.
- PRE_TENUTA - when the cylinder completes its strokes, the pressure inside the chambers is not the supply pressure; the cylinder starts to move when in the chambers there is more or less half of the supply pressure. If we just close the chamber at the end of the stroke, the pressure inside is lower than 10 bar. Before the seal test we open the valve that sends more air to the chamber we want to test, so that the pressure inside reaches 10 bar. The state in which we open this valve is called PRE_TENUTA. During the normal execution of the SEAL state, the cylinder passes through the PRE_TENUTA state two times; the first time is for the front chamber, while the second time is for the rear chamber.

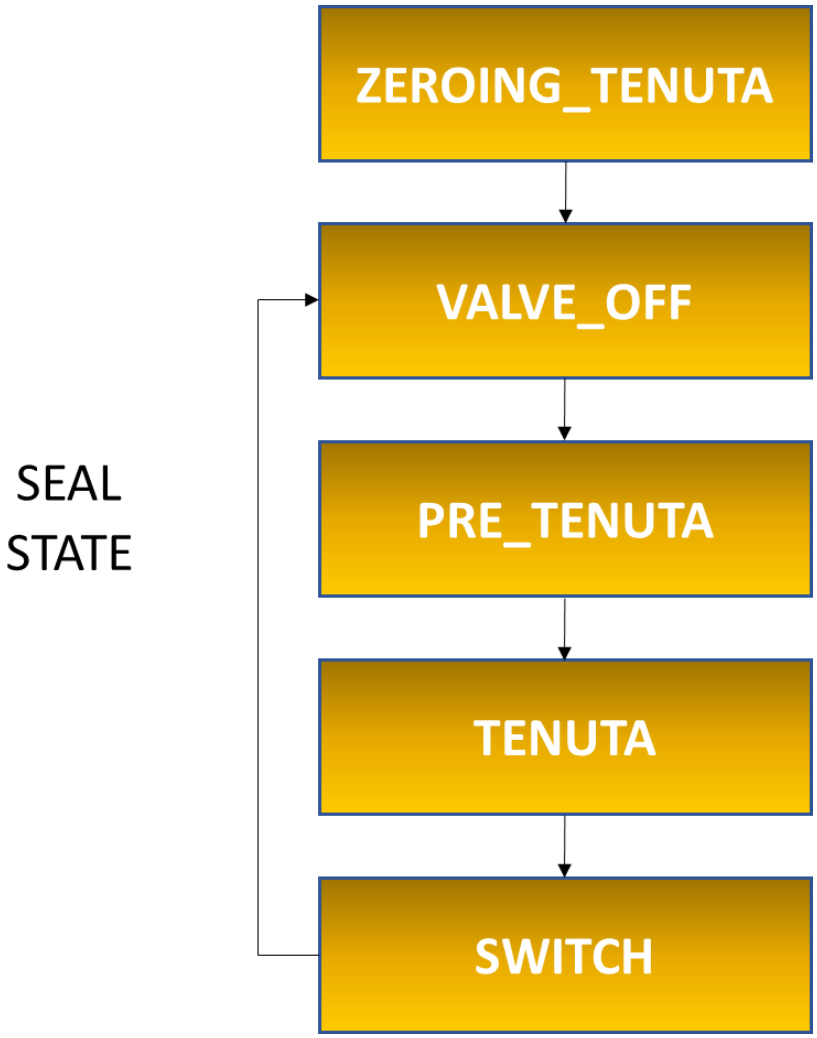

Figure 5-5: States of the SEAL state

- SWITCH - it is the state that separates the two seal tests. In this state we open the valve in order to have the piston rod fully extended and start the seal test for the rear chamber.

There are still two states that have to be taken into consideration. They have similar characteristics. These states are:

- PAUSA_USURA - During the USURA state it is possible to press the STOP button once in order to enter the PAUSA_USURA state, in which the cylinder freezes. To exit this state the user can press the STOP button for a second time, entering the IDLE_STATE state, or the WEAR button, entering the USURA state again and resuming the wear test exactly from where it was left.

- PAUSA_TENUTA - During the PRE_TENUTA, TENUTA and SWITCH states it is possible to press the STOP button once in order to enter the PAUSA_TENUTA state, in which the cylinder freezes. To exit this state the user can press the STOP button for a second time, entering the IDLE_STATE state, or the SEAL button, entering the ZEROING_TENUTA state and starting a new seal test from the beginning.

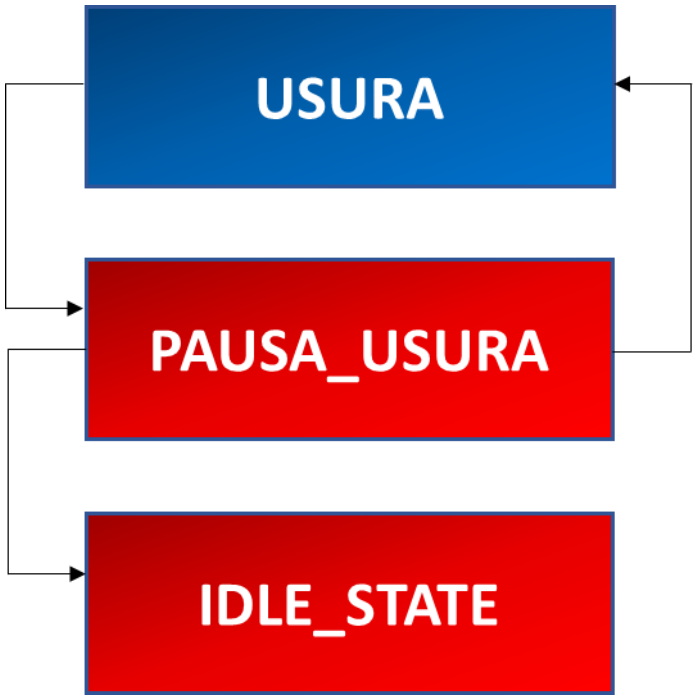

Figura 5-6: Flow chart for PAUSA_USURA

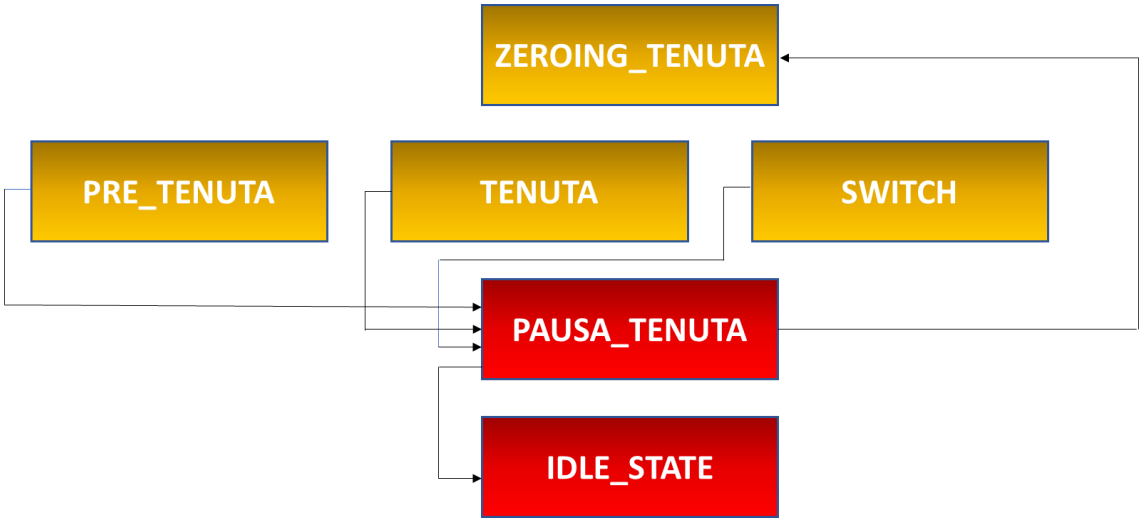

Figure 5-7: Flow chart for PAUSA_TENUTA

5.8 *CHECK_STATE()* FUNCTION

The purpose of the "check_state" function is to check the state in which the cylinder is, and according to this information perform some operations. It is declared after the *loop()* function, and the function prototype is:

```
1.  void check_state(int n_cil){
2.  }
```

Therefore, "check_state" needs the number of the cylinder "n_cil" as parameter. The "check_state" function is called in the loop() block right after the part where the "fineCorsa" variables are updated when PortKUpdated becomes true. Here is the piece of code where the function is invoked:

```
1.  for(i=0; i<N_CIL; i++)
2.    {check_state(i);}
```

The variable "i" is an integer variable defined before the *setup()* block. As we can see, the check_state function is called inside a for cycle. This function examines the first cylinder, then the second one.

The body of the check_state function is composed by a series of "if" conditions regarding most of the possible states of the cilynders. At each loop cycle, only one of these conditions can be satisfied, so only the part related to that particular state is executed.

Before examining each part one by one, some considerations have to be made about the valves. Digital pins 22 and 23 are used to connect the valve acting on the first cylinder, while digital pins 24 and 25 are used for the second valve. These pins are used as outputs. Inside the *cilindro* structure we define some variables related to the status of these pins. We create an array of two elements, "`int valvePin[2];`". The first element of the array "valvePin[0]" refers to the part of the valve that has to be electrically stimulated in order to have the compressed air entering the rear chamber, pushing the piston rod so that it becomes fully extended. The pins related to valvePin[0] are 22 and 24. When the voltage on these pins is high, current starts flowing through the solenoid that is on the side of the valve that pushes compressed air inside the rear chamber. The

pins related to the second element of the array valvePin[1] are, instead, 23 and 25. When these pins are set “HIGH” air enters the front chamber of the cylinders and the piston rod becomes fully retracted. The definition of these four pins as outputs is contained inside the *setup()* block. The portion of code is the following one:

```
1.   for(int n_cil=0; n_cil<N_CIL; n_cil++){
2.
3.      for(i=0;i<2;i++){
4.
5.        cilindri[n_cil].valvePin[i]=n_cil*2+i+22;
6.        pinMode(cilindri[n_cil].valvePin[i], OUTPUT);
7.        digitalWrite(cilindri[n_cil].valvePin[i], LOW);
8.
9.     }
10. }
```

As this fragment shows, the valves are initially closed.

As we said, not all the states are involved inside the check_state function. For example, no operation is done when the cylinder is in the states OFF, ROTTO and IDLE_STATE. We will now consider what happens inside the check_state function if the cylinder is in the ZEROING_USURA state:

```
1.  if(cilindri[n_cil].state==ZEROING_USURA){
2.     digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
3.     digitalWrite(cilindri[n_cil].valvePin[0], LOW);
4.     blinkWearLed(n_cil, 100);
5.
6.     if(digitalRead(cilindri[n_cil].reed[0])){
7.
8.       cilindri[n_cil].fineCorsa[0]=digitalRead(cilindri[n_cil].reed[0]);
9.       cilindri[n_cil].fineCorsa[1]=digitalRead(cilindri[n_cil].reed[1]);
10.
11.      digitalWrite(cilindri[n_cil].WEAR_LED, HIGH);
12.
13.      cilindri[n_cil].state=USURA;
14.
15.    }
16.
17.  }
```

This state is a preparation to the wear test, that we want to start with a fully retracted piston rod, so we set valvePin[1] high and valvePin[0] low. We also want the WEAR LED to blink in this state so we call the BlinkWearLed function, choosing 100 as the parameter T (the LED will be on and off for 0.1 seconds, alternately). In order to detect the moment when the piston rod is in the right position, we have an “if” condition on reed[0], because we want that particular Reed sensor to turn on. When this happens, the cylinder can enter the USURA state. Before that, we update the fineCorsa variables

with the values read from the Reed sensors, so fineCorsa[0] will become 1, and fineCorsa[1] will become 0. We also turn on the WEAR LED so that it stays on during the USURA state.

(N.B. the if conditions for each state are intertwined in such a way that operations related to a particular state, for example turning on the LED for the USURA state, are performed inside the if condition related to a previous state, in this case ZEROING_USURA. These will happen in many cases)

We will examine now the if condition related to the USURA state:

```
1.  if(cilindri[n_cil].state == USURA){
2.
3.      if(cilindri[n_cil].N == TARGET_N){
4.        cilindri[n_cil].N_tot += cilindri[n_cil].N ;
5.        cilindri[n_cil].N = 0;
6.        digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
7.        cilindri[n_cil].state = ZEROING_TENUTA;
8.        cilindri[n_cil].T0_blink=millis();
9.
10.      }
11.   }
```

The code responsible for making the piston rod go back and forth is written inside another function called “update_cil”, that will be explained later on. In this fragment, we only deal with the end of the USURA state, that happens when the established number of strokes is completed. This number is indicated inside our sketch with the name TARGET_N, and it is defined before the *setup()* block as an unsigned long int variable. In this portion we can also see:

- N, that is a long integer variable defined inside the *cilindro* structure, that counts the number of strokes completed since the beginning of that particular wear test. It is initialized to zero for all the cylinders inside the *setup()* block.
- N_tot, that is also a long integer variable defined inside the *cilindro* structure which refers to the total number of strokes completed since the Arduino board began running the program. We only take into consideration the strokes happening during the wear tests. It is initialized to zero for all the cylinders inside the *setup()* block.

So when the counter N reaches the target value indicated by TARGET_N, the wear test is over. N_tot is incremented by a number of strokes equal to TARGET_N, while counter N returns to the zero value in order to be ready for the next wear test. The WEAR LED is turned OFF and the cylinder enters the ZEROING_TENUTA state. T0_blink is initialized using the millis() function because the ZEROING_TENUTA state makes use of a blinking function.

The part of the check_state function related to ZEROING_TENUTA is the following one:

```
1.  if(cilindri[n_cil].state==ZEROING_TENUTA){
2.      digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
3.      digitalWrite(cilindri[n_cil].valvePin[0], LOW);
4.
5.        if(digitalRead(cilindri[n_cil].reed[0])){
6.
7.        cilindri[n_cil].fineCorsa[0]=digitalRead(cilindri[n_cil].reed[0]);
8.        cilindri[n_cil].fineCorsa[1]=digitalRead(cilindri[n_cil].reed[1]);
9.
10.       cilindri[n_cil].state=VALVE_OFF;
11.
12.       cilindri[n_cil].TENUTA_ANT=true;
13.       digitalWrite(cilindri[n_cil].valvePin[0], LOW);
14.
15.       digitalWrite(cilindri[n_cil].valvePin[1], LOW);
16.
17.       cilindri[n_cil].T0_valveoff=millis();
18.       cilindri[n_cil].T0_blink=millis();
19.
20.     }
21.     blinkSealLed(n_cil, 100);
22.   }
```

We want to perform first the seal test for the front chamber so we want to have a fully retracted piston rod. We do something identical with respect to what we have for the ZEROING_USURA state: we set valvePin[1] high, valvePin[0] low, then we wait until reed[0] becomes 1 meaning that the piston rod is in the correct position, then we update the fineCorsa variables. At this point the cylinder can enter the VALVE_OFF state, where we close both sides of the valve, so valvePin[0] and valvePin[1] are set to zero. We update a Boolean variable named “TENUTA_ANT”.

“TENUTA_ANT” is a Boolean variable that gives information about the particular chamber on which the seal test is performed. The value of this variable is true for the seal test on the front chamber, and becomes false when the seal test on the rear chamber is about to be performed.

In this case, we just exited from ZEROING_TENUTA, so the test on the front chamber is imminent and TENUTA_ANT has to be set to true.

Then we reset the T0_blink timer and also the T0_valveoff timer.

T0_valveoff is a long integer variable in which we store the time that has passed since the cylinder entered the VALVE_OFF state. When this time reaches a predefined target called TARGET_VALVEOFF, the cylinders exits from the VALVE_OFF state.

During ZEROING_TENUTA we also make the SEAL LED blink by calling the BlinkSealLed function using T=100.

The if condition related to the VALVE_OFF state is the following one:

```
1.  if(cilindri[n_cil].state==VALVE_OFF){
2.    if(millis()-cilindri[n_cil].T0_valveoff>TARGET_VALVEOFF){
3.     if(cilindri[n_cil].TENUTA_ANT){
4.      digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
5.      Serial.println("s" + String(n_cil));
6.
7.     }
8.     else{
9.      digitalWrite(cilindri[n_cil].valvePin[0], HIGH);
10.
11.    }
12.    cilindri[n_cil].T0_pretenuta=millis();
13.    cilindri[n_cil].T0_blink=millis();
14.
15.    cilindri[n_cil].state=PRE_TENUTA;
16.
17.   }
18.   blinkSealLed(n_cil, 200);
19.  }
```

During the VALVE_OFF state we make the SEAL LED blink using T=200.

When the time inside T0_valveoff reaches the target, we enter the PRE_TENUTA state. We know that the passage from VALVE_OFF to PRE_TENUTA happens twice during the normal execution of the SEAL state, the first time for the front chamber, then the second time for the rear chamber. If the cylinder is going through the first transition, it means that TENUTA_ANT is true, we want compressed air to enter the front chamber so we set valvePin[1] high. Vice versa, when the cylinder is going through the second passage, we check the value of TENUTA_ANT which is false, so in this case we need to set valvePin[0]

high. In both cases in the end we enter the PRE_TENUTA state, but first we have to reset two timers: T0_blink and T0_pretenuta.

T0_pretenuta is a long integer variable in which we store the time that has passed since the cylinder entered the PRE_TENUTA state. When this time reaches a predefined target called TARGET_PRE_T, the cylinders exits from the PRE_TENUTA state.

During the passage to PRE_TENUTA that regards the front chamber, we also send a string to Raspberry Pi that informs that a seal test is about to be performed. When Raspberry Pi receives “s0” knows that it has to create a file to store the pressure values for the first cylinder. Similarly, “s1” is the string that informs Raspberry Pi that a file for the second cylinder has to be created.

Now we will consider the if condition related to the PRE_TENUTA state:

```
1.  if(cilindri[n_cil].state==PRE_TENUTA){
2.      if(millis()-cilindri[n_cil].T0_pretenuta>TARGET_PRE_T){
3.        cilindri[n_cil].T0=millis();
4.        cilindri[n_cil].T0_blink=millis();
5.        cilindri[n_cil].state=TENUTA;
6.
7.        if(cilindri[n_cil].TENUTA_ANT){
8.          digitalWrite(cilindri[n_cil].valvePin[1], LOW);
9.
10.       }
11.       else{
12.         digitalWrite(cilindri[n_cil].valvePin[0], LOW);
13.
14.        }
15.     }
16.      blinkSealLed(n_cil, 200);
17.   }
```

During the PRE_TENUTA state we make the SEAL LED blink using T=200.

When the time inside T0_PRETENUTA reaches the target, we enter the TENUTA state. We know that the passage from PRE_TENUTA to TENUTA happens twice during the normal execution of the SEAL state, the first time for the front chamber, then the second time for the rear chamber. If the cylinder is going through the first transition, valvePin[1] has to be set low in order to have the valve closed. Vice versa, we set to low valvePin[0].

Timer T0_blink is reset because we also call the BlinkSealLed function in the TENUTA state. We also reset another timer called T0.

T0 is a long integer variable in which we store the time that has passed since the cylinder entered the TENUTA state. When this time reaches a predefined target called TARGET_T, the cylinders exits from the TENUTA state.

The fragment of code that is showed next is related to the if condition on the TENUTA state:

```
1.  if(cilindri[n_cil].state == TENUTA){
2.      if(cilindri[n_cil].TENUTA_ANT){
3.        if((millis()-cilindri[n_cil].T0)>TARGET_T){
4.          float p =(float)(analogRead(cilindri[n_cil].pressSens)-
    ANALOG_TRANSLATION)*ANALOG_TO_BAR;
5.          if(p < BROKEN_PRESSURE){
6.
7.             cilindri[n_cil].tenutaOK[0]=false;
8.
9.          }
10.
11.         cilindri[n_cil].state=SWITCH;
12.
13.         digitalWrite(cilindri[n_cil].valvePin[0], HIGH);
14.
15.         cilindri[n_cil].TENUTA_ANT=false;
16.         cilindri[n_cil].T0_blink=millis();
17.
18.       }
19.     }
20.     else{
21.       if((millis()-cilindri[n_cil].T0)>TARGET_T){
22.         float p =(float)(analogRead(cilindri[n_cil].pressSens)-
    ANALOG_TRANSLATION)*ANALOG_TO_BAR;
23.
24.         if(p < BROKEN_PRESSURE){
25.
26.            cilindri[n_cil].tenutaOK[1]=false;
27.
28.         }
29.         if(cilindri[n_cil].tenutaOK[0]==true && cilindri[n_cil].tenutaOK[1]==tr
    ue)  {
30.         digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
31.         cilindri[n_cil].state = ZEROING_USURA;
32.
33.         }
34.         else{
35.           digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
36.           digitalWrite(cilindri[n_cil].SELECT_LED, LOW);
37.           digitalWrite(cilindri[n_cil].BROKEN_LED, HIGH);
38.           cilindri[n_cil].state = ROTTO;
39.         }
40.       }
41.
42.     }
43.     blinkSealLed(n_cil, 500);
44.   }
```

During the TENUTA state the BlinkSealLed function is called with T=500.

There are two parts, one for the seal test on the front chamber (inside the if condition on TENUTA_ANT) and one for the seal test on the rear chamber (inside the else condition, that starts at line 20 of this fragment). In both cases, when T0 reaches TARGET_T and the end of the seal test is reached, we check the pressure value p. If p is lower than a value called BROKEN_PRESSURE, the seal test has not been passed. BROKEN_PRESSURE is defined through the #define directive before the *setup()* function. We decided to consider the seal test failed when the pressure becomes lower than 0.5 bar, so we write:

```
#define BROKEN_PRESSURE 0.5
```

If p is lower than BROKEN_PRESSURE for a specific chamber, the value of tenutaOK becomes false for that chamber. The definition of tenutaOK takes place inside the *cilindro* structure by means of the line "`bool tenutaOK[2];`". It is an array of Boolean variables. The first component "tenutaOK[0]" becomes false when the seal test has not been passed for the front chamber, while "tenutaOK[1]" refers to the other chamber. Both components are set to true inside the *setup()* function:

```
1.  for(int n_cil=0;n_cil<N_CIL;n_cil++){
2.
3.     cilindri[n_cil].tenutaOK[0]=true;
4.     cilindri[n_cil].tenutaOK[1]=true;
5.
6.   }
```

The presence of this flag is due to the fact that we want to perform the test on both chambers, even if the test on the first chamber fails. Then when the cylinder is at the end of the second test we need to verify if both flags are still true. If it is so, the SEAL LED is turned off and the cylinder can enter the ZEROING_USURA state, starting a new wear test; otherwise, the cylinder enters the ROTTO state, the BROKEN LED is turned on while the SELECT LED and the SEAL LED are turned off.

At the end of the seal test for the front chamber, the cylinder enters the SWITCH state. We need to perform the test on the rear chamber, so valvePin[0] is set to high, TENUTA_ANT becomes false and T0_blink is reset, because the SEAL_LED has to blink even in the SWITCH state.

Now we will consider the if condition related to the SWITCH state:

```
if(cilindri[n_cil].state==SWITCH){
    if(cilindri[n_cil].fineCorsa[1]){
       cilindri[n_cil].state=VALVE_OFF;

       digitalWrite(cilindri[n_cil].valvePin[0], LOW);

       digitalWrite(cilindri[n_cil].valvePin[1], LOW);

       cilindri[n_cil].T0_valveoff=millis();
       cilindri[n_cil].T0_blink=millis();

    }
    blinkSealLed(n_cil, 100);
  }
```

In this state, the SEAL LED blinks, being on and off alternately for T=100.

We need to check when fineCorsa[1] becomes true in order to be sure that the piston rod has become fully extended and the cylinder is ready for a seal test on the rear chamber. If fineCorsa[1] is true, the cylinder enters the VALVE_OFF state, the valve becomes closed and the two timers T0_valveoff and T0_blink are reset.

The remaining part of the check_state function is the following one:

```
if(cilindri[n_cil].state == PAUSA_USURA){
  blinkStopLed(n_cil, 100);

  }

  if(cilindri[n_cil].state == PAUSA_TENUTA){
  blinkStopLed(n_cil, 500);

  }
```

In the states of PAUSA_USURA and PAUSA_TENUTA we make the STOP LED blink. We choose T=100 for PAUSA_USURA and T=500 for PAUSA_TENUTA, two different values so that the user can understand in which state the cylinder is by recognizing the blinking frequency of the STOP LED, which is slower in the case of PAUSA_TENUTA.

5.9 *UPDATE_CIL()* FUNCTION

The purpose of the “update_cil” function is to make possible the wear test of the cylinders in the USURA state, by making the piston rod go back and forth. This function

is called inside the *loop()* block after the check_state function by means of this piece of code:

```
for(j=0; j<N_CIL; j++)
  {update_cil(j);}
```

The variable "j" is an integer variable defined before the *setup()* block. As we can see, the update_cil function is called inside a for cycle so it takes into consideration first the first cylinder, then the second one.

Here is the code for the function:

```
void update_cil(int n_cil){

  if(cilindri[n_cil].state == USURA && cilindri[n_cil].fineCorsa[0]){

        cilindri[n_cil].fineCorsa[0]=0;

        if(cilindri[n_cil].N <TARGET_N-1){

          digitalWrite(cilindri[n_cil].valvePin[0],HIGH);
          digitalWrite(cilindri[n_cil].valvePin[1],LOW);

        };

        cilindri[n_cil].N++;

  }

  if(cilindri[n_cil].state == USURA && cilindri[n_cil].fineCorsa[1]){

        cilindri[n_cil].fineCorsa[1]=0;

        if(cilindri[n_cil].N <TARGET_N){

           digitalWrite(cilindri[n_cil].valvePin[0],LOW);
           digitalWrite(cilindri[n_cil].valvePin[1],HIGH);

        };

        cilindri[n_cil].N++;

  }

}
```

If the cylinder is not in the USURA state, no operation will be performed. In this state we need to check the fineCorsa variables. If they are true, it means that the piston rod is at the end of the stroke so it can be pushed to other side. So if fineCorsa[0] is true, it is immediately set to false, we increment the value of the counter N (number of strokes of the specific wear test) and we set valvePin[0] to high and valvePin[1] to low in order to start another stroke and have a piston rod fully extended. Then fineCorsa[1] becomes

true when the stroke is completed, so we verify the other if condition. We set to false fineCorsa[1] than we start another stroke towards the opposite side by setting valvePin[0] to low and valvePin[1] to high, in order to have at the end a fully retracted piston rod.

There are two if conditions on the counter N in order to make sure that nothing is done to the valves if N is out of the proper range.

5.10 *UPDATE_BUTT()* FUNCTION

There is a last function inside our sketch that has to be taken into consideration. The name of the function is “update_butt”.

This function is the last function called inside the *loop()* block, right before the default value -1 is assigned to the “butt” variable. The update_butt function receives no parameter and does not return any value. The body of the function is composed by a series of if conditions that examine the value inside the “butt” variable.

As we have already mentioned, when a button is pressed the PCF sends an interrupt and by examining the voltage on the pins of the PCF it is possible to understand which button has been pressed. The function GetSwitchPress is responsible for changing the value of the “butt” variable, depending on the pin of the PCF and consequently on the button too. Then inside the update_butt function it is decided what has to be done according to the particular value of “butt”. There are twelve buttons, so twelve different values for “butt” have to be considered.

We will now take into account the code for the first column of buttons on the cover of the box, the SELECT buttons.

```
1.  if(butt==10000) //ON OFF TOTALE
2.    {
3.      for(int n_cil=0; n_cil<N_CIL; n_cil++){
4.        if(cilindri[n_cil].state==OFF){
5.          cilindri[n_cil].state = IDLE_STATE;
6.          digitalWrite(cilindri[n_cil].SELECT_LED, HIGH);
7.          digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
```

```
8.
9.          }
10.       else{
11.         cilindri[n_cil].N_tot += cilindri[n_cil].N ;
12.         cilindri[n_cil].N = 0;
13.         cilindri[n_cil].state = OFF;
14.         digitalWrite(cilindri[n_cil].STOP_LED, LOW);
15.         digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
16.         digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
17.         digitalWrite(cilindri[n_cil].SELECT_LED, LOW);
18.         digitalWrite(cilindri[n_cil].BROKEN_LED, LOW);
19.
20.         }
21.     }
22.   }
23.
24.   if(butt==11000) //ON OFF UN CILINDRO
25.   {
26.       if(cilindri[0].state==OFF){
27.         cilindri[0].state = IDLE_STATE;
28.         digitalWrite(cilindri[0].SELECT_LED, HIGH);
29.         digitalWrite(cilindri[0].STOP_LED, HIGH);
30.
31.         }
32.       else{
33.         cilindri[0].N_tot += cilindri[0].N ;
34.         cilindri[0].N = 0;
35.         cilindri[0].state = OFF;
36.         digitalWrite(cilindri[0].STOP_LED, LOW);
37.         digitalWrite(cilindri[0].SEAL_LED, LOW);
38.         digitalWrite(cilindri[0].WEAR_LED, LOW);
39.         digitalWrite(cilindri[0].SELECT_LED, LOW);
40.
41.         }
42.
43.   }
44.
45.    if(butt==12000) //ON OFF UN CILINDRO
46.   {
47.       if(cilindri[1].state==OFF){
48.         cilindri[1].state = IDLE_STATE;
49.         digitalWrite(cilindri[1].SELECT_LED, HIGH);
50.         digitalWrite(cilindri[1].STOP_LED, HIGH);
51.
52.         }
53.       else{
54.         cilindri[1].N_tot += cilindri[1].N ;
55.         cilindri[1].N = 0;
56.         cilindri[1].state = OFF;
57.         digitalWrite(cilindri[1].STOP_LED, LOW);
58.         digitalWrite(cilindri[1].SEAL_LED, LOW);
59.         digitalWrite(cilindri[1].WEAR_LED, LOW);
60.         digitalWrite(cilindri[1].SELECT_LED, LOW);
61.
62.         }
63.
64.   }
```

The value 10000, stored inside the variable butt, refers to the global SELECT button, that acts on all the cylinders simultaneously, enabling a single cylinder if it is disabled and vice versa. The value 11000 is related to the SELECT button for the first cylinder, while the value 12000 refers to the SELECT button for the second cylinder. In this portion of

code it is possible to see that what is done for each of these three cases is almost identical. The only difference is that for the global button there is a for cycle in order to process all the cylinders, while for the other buttons we just act on the fields of cilindri[0] or cilindri[1], depending on the button and the cylinder. So, when we will discuss the code for the other buttons, it will be just sufficient to consider the code for the global buttons.

Now we will explain the behavior of the SELECT button. If the cylinder is in the OFF state when the SELECT button is pressed, the cylinder enters the IDLE_STATE state and the SELECT LED and STOP LED are turned on. Otherwise, if the cylinder is in any other state, it enters the OFF state and all five LEDs are turned off. In this case, it is possible that N, the counter of the strokes completed in a wear test, is different from zero, for example in the case we press the SELECT button when the cylinder is in the USURA state in the midst of a wear test. N_tot is updated by adding the value of N, that returns to zero so that the next time a wear test will be performed it will start from the beginning.

The code related to the global STOP button, exactly equivalent to the code for the other STOP buttons, is the following one:

```
1.  if(butt==300) //PAUSA STOP TOTALE
2.   {
3.     for(int n_cil=0; n_cil<N_CIL; n_cil++){
4.
5.          if(cilindri[n_cil].state==USURA){
6.            cilindri[n_cil].state = PAUSA_USURA;
7.            cilindri[n_cil].T0_blink=millis();
8.            digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
9.
10.           cilindri[n_cil].valvePin_prestop[0]=digitalRead(cilindri[n_cil].valv
    ePin[0]);
11.           cilindri[n_cil].valvePin_prestop[1]=digitalRead(cilindri[n_cil].valv
    ePin[1]);
12.
13.           digitalWrite(cilindri[n_cil].valvePin[0], LOW);
14.           digitalWrite(cilindri[n_cil].valvePin[1], LOW);
15.          }
16.         else if(cilindri[n_cil].state==PRE_TENUTA || cilindri[n_cil].state==TE
    NUTA || cilindri[n_cil].state==SWITCH ){
17.
18.           cilindri[n_cil].state = PAUSA_TENUTA;
19.           cilindri[n_cil].T0_blink=millis();
20.           digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
21.
22.           digitalWrite(cilindri[n_cil].valvePin[0], LOW);
23.           digitalWrite(cilindri[n_cil].valvePin[1], LOW);
24.
25.         }
26.          else if(cilindri[n_cil].state == PAUSA_USURA){
```

```
27.           cilindri[n_cil].state = IDLE_STATE;
28.           digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
29.
30.           digitalWrite(cilindri[n_cil].valvePin[0], LOW);
31.           digitalWrite(cilindri[n_cil].valvePin[1], LOW);
32.
33.           cilindri[n_cil].N_tot += cilindri[n_cil].N;
34.           cilindri[n_cil].N = 0;
35.
36.          }
37.          else if(cilindri[n_cil].state == PAUSA_TENUTA){
38.            cilindri[n_cil].state = IDLE_STATE;
39.            digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
40.
41.           digitalWrite(cilindri[n_cil].valvePin[0], LOW);
42.           digitalWrite(cilindri[n_cil].valvePin[1], LOW);
43.
44.            }
45.
46.      }
47.   }
```

When the value of butt is 300, the routine for the global STOP button is executed. The STOP button actually acts only during some states. There are four if conditions and four groups of operations. When the STOP button is pressed, the states in which the cylinder can enter are PAUSA_USURA, PAUSA_TENUTA and IDLE_STATE, so in each of the four groups, valvePin[0] and valvePin[1] are set to low in order to stop the cylinder.

If the cylinder is in the USURA state, it enters PAUSA_USURA. The WEAR LED is turned off and the timer T0_blink is reset, because in PAUSA_USURA the STOP LED is blinking. The voltage on the pins of the valve is read and stored inside some variables called valvePin_prestop. These variables are defined inside the *cilindro* structure by means of the line "`int valvePin_prestop[2];`", they are initialized to zero in the *setup()* function and they are used to store the condition of the valve when the wear test is paused, so that the same condition can be restored if the user decides to resume the wear test. The wear test can continue as if nothing actually happened.

After the level of the pins of the valve is saved inside the valvePin_prestop array, the valve can be closed so we set to low the valvePin variables.

If the cylinder is in the PRE_TENUTA, TENUTA and SWITCH states, it enters the PAUSA_TENUTA state if the STOP button is pressed. The SEAL LED is turned off and the timer T0_blink is reset, because PAUSA_TENUTA is a state with a blinking LED. The valve is closed.

If the cylinder is in the PAUSA_USURA state, it enters the IDLE_STATE state. The STOP_LED is turned on and the valve is closed. The value of N, most likely different from zero because the cylinder entered the PAUSA_USURA state from the USURA state, is added to N_tot and then reset so that the next wear test will start from the beginning.

The cylinder enters the IDLE_STATE state also if it is in the PAUSA_TENUTA state. The STOP_LED is turned on and the valve is closed in this case too.

We will now consider how the WEAR button works by looking at the related portion of code:

```
1.  if (butt==200)   //USURA TOTALE
2.    {
3.      for(int n_cil=0; n_cil<N_CIL; n_cil++){
4.        if(cilindri[n_cil].state == PAUSA_USURA){
5.          cilindri[n_cil].state = USURA;
6.          digitalWrite(cilindri[n_cil].STOP_LED, LOW);
7.
8.              digitalWrite(cilindri[n_cil].valvePin[0], cilindri[n_cil].valvePin
    _prestop[0]);
9.              digitalWrite(cilindri[n_cil].valvePin[1], cilindri[n_cil].valvePin
    _prestop[1]);
10.
11.             digitalWrite(cilindri[n_cil].WEAR_LED, HIGH);
12.         }
13.
14.       else if(cilindri[n_cil].state == IDLE_STATE){
15.
16.             cilindri[n_cil].state = ZEROING_USURA;
17.
18.             digitalWrite(cilindri[n_cil].STOP_LED, LOW);
19.         }
20.     }
21.   }
```

When the value of butt is 200, the routine for the global WEAR button is executed.

If the cylinder is in the IDLE_STATE state, it enters the ZEROING_USURA state when the WEAR button is pressed. The STOP LED is turned off. A new wear test is about to start.

If the cylinder is in the PAUSA_USURA state, it enters the USURA state. The STOP LED is turned off and the WEAR LED is turned on. The values stored inside valvePin_prestop are used to rewrite the content of the valvePin variables, in order to resume the wear test as if the pause never happened.

Finally, here is the code for the global SEAL button:

```
1.  if(butt==100)  //TENUTA TOTALE
2.  {
3.    for(int n_cil=0; n_cil<N_CIL; n_cil++){
4.      if(cilindri[n_cil].state==USURA || cilindri[n_cil].state == IDLE_STATE){
5.        digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
6.        digitalWrite(cilindri[n_cil].STOP_LED, LOW);
7.        cilindri[n_cil].state = ZEROING_TENUTA;
8.
9.
10.   }
11.   else if(cilindri[n_cil].state == PAUSA_TENUTA){
12.       cilindri[n_cil].state = ZEROING_TENUTA;
13.       digitalWrite(cilindri[n_cil].STOP_LED, LOW);
14.
15.     }
16.  }
17. }
```

In this case, the value of butt is 100. If the cylinder is in the USURA, IDLE_STATE or PAUSA_TENUTA state, it enters the ZEROING_TENUTA state and a new seal test starts from the beginning. The WEAR LED and STOP LED are turned off. In this case, during the transition from USURA to ZEROING_TENUTA, the value of N is not reset. This means that when a wear test will start after the end of the seal test, the cylinder will not have to complete a number TARGET_N of strokes in order to finish the wear test, but only the remaining strokes will be necessary.

6 RASPBERRY PI

Raspberry Pi is a single-board computer, namely a general-purpose computer implemented on a single circuit board. It contains a microprocessor, memory, input/output (I/O) and other features commonly required by any functional computer.

The particular type of Raspberry Pi we used in our test bench is a Raspberry Pi 3B +. This board features a quad core 64-bit processor clocked at 1.4 GHz, 1 GB SRAM, 40 GPIO pins, integrated Wi-Fi 2.4 GHz and 5 GHz, 300 Mbps Ethernet speed, 4.2 Bluetooth.

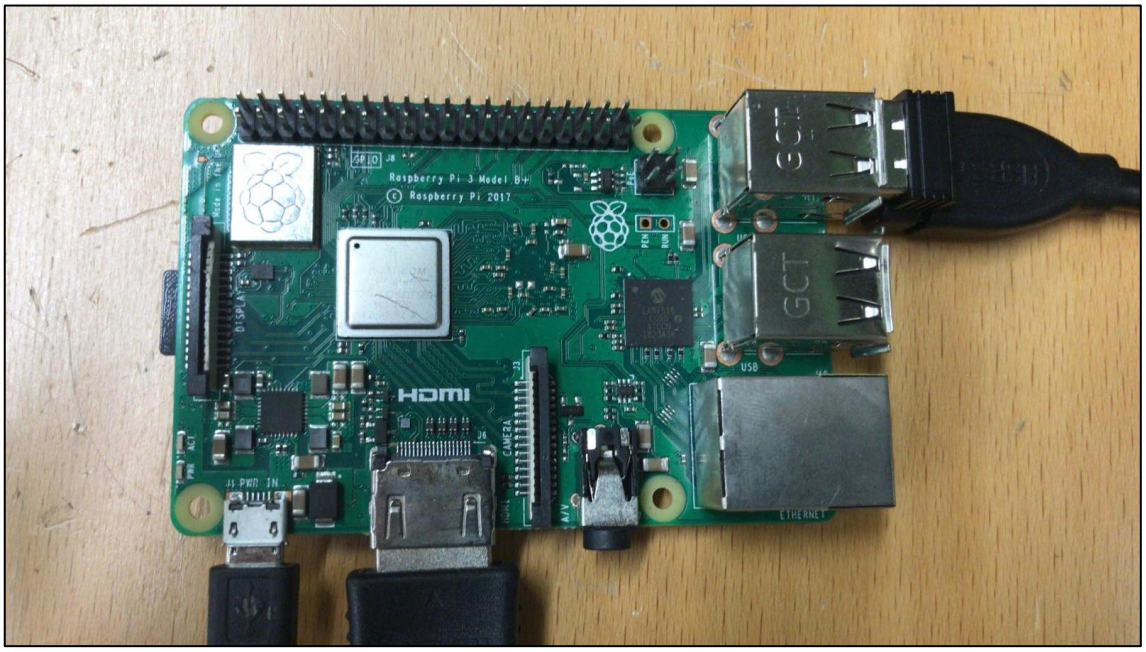

Figure 6-1: Raspberry Pi 3B+

Raspberry Pi also has four USB 2.0 ports. We used one of them to make the serial communication possible between Raspberry Pi and Arduino by means of a USB cable. Another USB port was used to connect wireless mouse and keyboard. In order to see the screen of the Raspberry Pi, we plugged into the Raspberry Pi an HDMI-equipped monitor using a standard HDMI cable. The device is powered by 5V micro USB, and the use of a 2.5 A power supply is recommended [9].

The operating system we installed by means of an SD card on the Raspberry Pi board is named Raspbian. Raspbian is a free operating system optimized for the Raspberry Pi hardware and based on Debian; the type of kernel is Linux.

6.1 THE PYTHON CODE

The language we decided to use for our program in Raspberry Pi is Python. The code we wrote for our Python program is the following one:

```
1.  #!/usr/bin/python2.7
2.
3.  import serial
4.  import time
5.  import csv
6.  import os
7.  import datetime
8.
9.  ser=serial.Serial('/dev/ttyACM0', 115200)
10. time.sleep(2)
11. ser.flushInput()
12. ser.write(b'1')
13.
14. while True:
15.     while ser.inWaiting()==0:
16.         pass
17.     data= ser.readline()
18.     print(data)
19.     strindata=str(data)
20.
21.     if strindata.startswith("00,"):
22.         csvwriter0.write(strindata[3:])
23.         #csvwriter0.write('\n')
24.         csvwriter0.flush()
25.         os.fsync(csvwriter0.fileno())
26.     elif strindata.startswith("11,"):
27.         csvwriter1.write(strindata[3:])
28.         #csvwriter1.write('\n')
29.         csvwriter1.flush()
30.         os.fsync(csvwriter1.fileno())
31.     elif strindata.startswith("s0"):
32.         #s="/var/www/html/CYL1/"+datetime.datetime.now().strftime("%Y_%m_%d-
    %H.%M")+".txt"
33.         s="/var/www/html/CYL1/"+datetime.datetime.now().strftime("%Y-%m-
    %d")+".csv"
34.         csvwriter0=open(s,"wb")
35.     elif strindata.startswith("s1"):
36.         s="/var/www/html/CYL2/"+datetime.datetime.now().strftime("%Y-%m-
    %d")+".csv"
37.         csvwriter1=open(s,"wb")
38.
```

The first shebang line was added because we installed the Python 3 version on our Raspberry Pi, but we wanted to be sure that no problem would arise even in the presence of some code written according to the standards of the Python 2 version.

We imported a series of modules. The module that had to be imported to allow the serial communication between Arduino and Raspberry is "serial". The line used to open the serial port ttyACM0, the port to which the Arduino Mega board is connected, is "`ser=serial.Serial('/dev/ttyACM0', 115200)`". The speed of the serial communication is 115200 baud, which means that the serial port is capable of transferring a maximum of 115200 bits per second. The same baud rate was chosen in Arduino.

The sleep() function suspends the execution of the program for a given number of seconds, in our case two seconds. In Python the module "time" that we imported provides several useful functions to handle time-related tasks.

To clear the input serial buffer we write "`ser.flushInput()`" . Then with the next line we write the byte "1" to the serial port and we send it to Arduino. This is done because the Arduino power up is faster than the one of the Raspberry Pi, so we wanted Arduino to wait for a signal (in our case a '1') from Raspberry Pi to know when the Raspberry Pi is ready and then start the other operations. The first lines of the *setup()* block inside the Arduino sketch are:

```
1.  Serial.begin(115200);
2.
3.      while(!Serial.available()){
4.
5.      }
6.
7.      while(1){
8.        if(Serial.read()== '1') break;
9.        delay(5);
10.     }
```

Serial.begin is used to start the serial communication, specifying the 115200 baud rate. By means of the first while loop, Arduino waits until there are some bytes ready for reading from the serial port. In order to go on with the execution of the sketch and exit from the while(1) loop, Arduino has to read the '1' that was sent by the Raspberry Pi. Arduino checks if the '1' has been sent every 5 milliseconds, thanks to the delay() function.

After the first setup instructions inside the Python program, there is a “while: True” infinite loop that will be executed endlessly and that contains the rest of the lines of program.

The inWaiting() function (old name for the Python 3 function in_waiting()) gets the number of bytes in the input buffer. If this number is zero, by means of the pass statement no command is executed. If the return value of the inWaiting() function is different from 0, it means that something is ready to be read. The content of the input buffer is read by means of the readline() function, that reads and returns one line from the stream, and it is stored inside a variable named “data”. The content of the variable data is printed on the terminal window of the Raspberry Pi and then converted into a string by means of str(). The result is stored inside “strindata”.

Then we have a series of if and elif conditions in which we apply the method startswith() on strindata. This method returns True if the string starts with the specified value, otherwise False.

If the data sent by Arduino starts with “s0” or “s1”, it means that respectively the first cylinder or the second cylinder entered the PRE_TENUTA state and a new seal test is about to begin. When this occurrence happen, Raspberry has to create a file for the cylinder that is being tested in order to store the results of the seal test. The files created are CSV files.

The acronym CSV stands for Comma Separated Values. CSV is a simple file format used to arrange and store tabular data, such as databases or spreadsheets, according to a specific structure. A CSV file stores tabular data (text and numbers, printable ASCII and UNICODE characters) in plain text. Each line of the CSV file is a data record. Each record is composed by one or more fields, separated by commas. The presence of the comma as a field separator is the reason behind the name for this file format. CSV files are a good choice when there is the need to handle large amounts of data that also need to be imported or used in other programs. In order to work on CSV files in Python, there is a module named “csv”.

To create the files we specify the folder paths, so the files will be stored inside the folder "/var/www/html/CYL1" in the case of the first cylinder and inside the folder "/var/www/html/CYL2" in the case of the second cylinder. Then the name of the file will be in the format "Year – month – day". We are not specifying the time of the day for the seal tests, since we suppose that there will not be more than one seal test per day for each cylinder. This is due to the fact that we suppose that TARGET_N will be around 60000 (30000 cycles) in a real wear test, and each seal test for each chamber will last about half an hour. In case the user decides to perform more than a seal test in one day, the content of the files is simply overwritten.

In order to give proper names to the files, we need to import the datetime module. We apply the now() method on the datetime class of the datetime module, having as a result a datetime object containing the current date and time. Then we apply the strftime() method to convert the datetime object into a string. To choose the string format we write as argument "%Y-%m-%d" [10]. Then we add to the name of the file ".csv" because we are creating a CSV file.

We open this file temporarily named "s" using the open() built-in function, that maps a physical file to a built-in file object. The open() method returns a file object that is assigned to two different variables, named "csvwriter0" in the case of the first cylinder and "csvwriter1" for the second cylinder. The parameter "wb" refers to the opening mode: in this case, the file is opened for writing only in binary format. Choosing mode "w" a file is created if it does not exist, or truncated if it exists.

After the file has been created during the PRE_TENUTA state, Arduino sends a series of strings with comma-separated values during the TENUTA state. If those strings start with "00," they are saved inside the first file applying the method write() to the file object cswriter0, otherwise if the strings start with "11," write() is applied to cswriter1. Before saving the strings, we remove the first three characters of each string by means of "strindata[3:]", because we do not want to store "00," and "11,".

Writing to disk is a slow operation, so many programs store up writes into large blocks which they write all at the same time. This is named "buffering" and Python does it

automatically when a file is opened. When we write to a file, we are actually writing to a "buffer" in memory. When the buffer becomes full, Python will automatically write it to disk. In order to tell Python to write everything in the buffer to disk immediately and flush the internal buffer we use “cswriter0.flush()” or “cswriter1.flush()”. However, the operating system buffers writes as well, so we need to use the fsync() method imported from the module “os”. This method, applied to the file descriptor of a file, forces the writing of the buffer of the file. By using “f.fileno()” with f being csvwriter0 or csvwriter1, we obtain the file descriptors for the files. Then we do os.fsync(f.fileno()), ensuring that all internal buffers associated with f are written to disk.

6.2 RASPBERRY PI AS WEB SERVER

In our test bench, we used the Raspberry Pi board as a web server, to host a website where it is possible to find information about the results of the seal tests performed on the cylinders. In order to explain what a web server is, some concepts have to be introduced.

6.2.1 CONCEPTS OF WEB SERVER AND PHP

On the Web, the communication (intended as an exchange of files) between different terminals, takes place by means of some transmission systems named protocols, among which one of the most famous is the HTTP protocol (Hyper Text Transfer Protocol). This protocol works in such a way that the user requests access to some resources (input), typing an URL or clicking on a link. These resources are then sent (output) when they are available; generally under the form of web pages or images.

With this procedure, the user simply communicates to the browser, installed on his computer (client), the path to be taken in order to obtain the searched information, hosted on a different computer (server). Therefore, a server is a computer that contains and provides files.

With the specific term “Web Server” we refer to a physical computer (in our case Raspberry Pi) that has the function to host and provide web sites, composed by static pages (containing only HTML code) or dynamic pages (containing PHP code, as in our case) [11]. The distinction between these two types of pages is important to understand the particular role of the Web Server.

A simple HTML page does not require any particular intervention from the Web Server, because the code is already interpreted by the browser installed on the client computer; the situation changes in the presence of PHP code (dynamic pages) used to generate dynamically HTML code. It is in this second case that the intervention and mediation of a Web server is necessary.

We can also define a Web Server as a software installed inside a computer (server), with the purpose of elaborating web pages and generating content dynamically. We used one of the most widespread Web servers: Apache HTTP Server, developed by Apache Software Foundation.

PHP is a server-side programming language. While codes based on client-side programming languages such as Javascript are executed on the client computer, the PHP code is launched on the server. When a button connected to a link is clicked on a website, a request with some arguments is sent to the server, then the requested PHP code with those particular arguments is launched on the server and a response is sent back, in the form of a web page or other things.

As an example to understand the difference between client-side and server-side and the reason why PHP should be used, we can consider our website and the situation when the user wants to login, inserting username and password. In order to check if the typed access credentials are correct, they have to be compared with the ones in the database. Sending the whole database to the client and allowing this comparison on the user’s PC would not be a good a choice, because in this way the user would know the credentials of all the other users. Instead, what actually happens is that the username and the password typed by the user are sent to the server, that needs to launch a specific PHP program. That program uses the database containing usernames and passwords and

checks if the access credentials that were sent by the client are present in the database. If they are, the program returns the main menu page, otherwise it sends the user back to the login page.

6.2.2 THE WEBSITE

All the content and the webpages of the website are stored inside the Raspberry Pi web server, in particular inside a default folder named "/var/www/html". The default web page is a PHP page, located at "/var/www/html/index.php". This default web page is served when the user browses "http://localhost/" on the Raspberry Pi itself or the Raspberry Pi's IP address instead of "localhost" from another computer on the same subnetwork of the Pi.

For example, the default page that the user sees entering the website is the one represented in this screenshot:

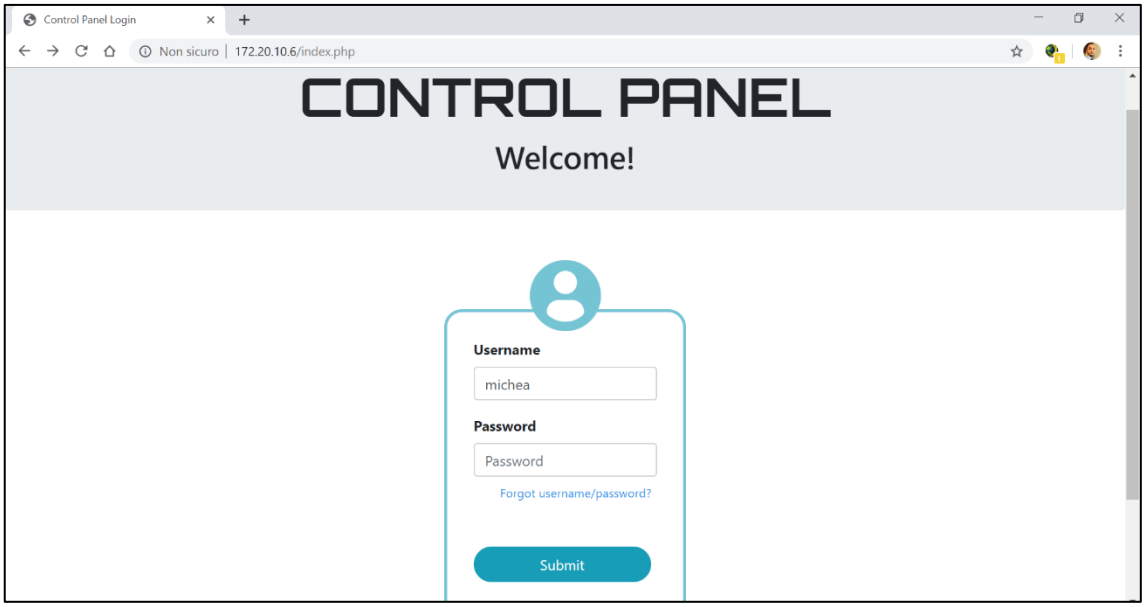

Figure 6-2 Log-in page for the website

This is a screenshot taken from a computer connected to the same subnetwork of the Raspberry Pi. Both the devices shared the same Wi-fi connection. “172.20.10.6”, the address that can be seen in the figure inside the address bar, is the IP address of the Raspberry Pi, found typing “ifconfig” on its terminal. In this default page named “index.php” the user can enter the access credentials.

Now we are interested in seeing the page that shows the results of the seal tests:

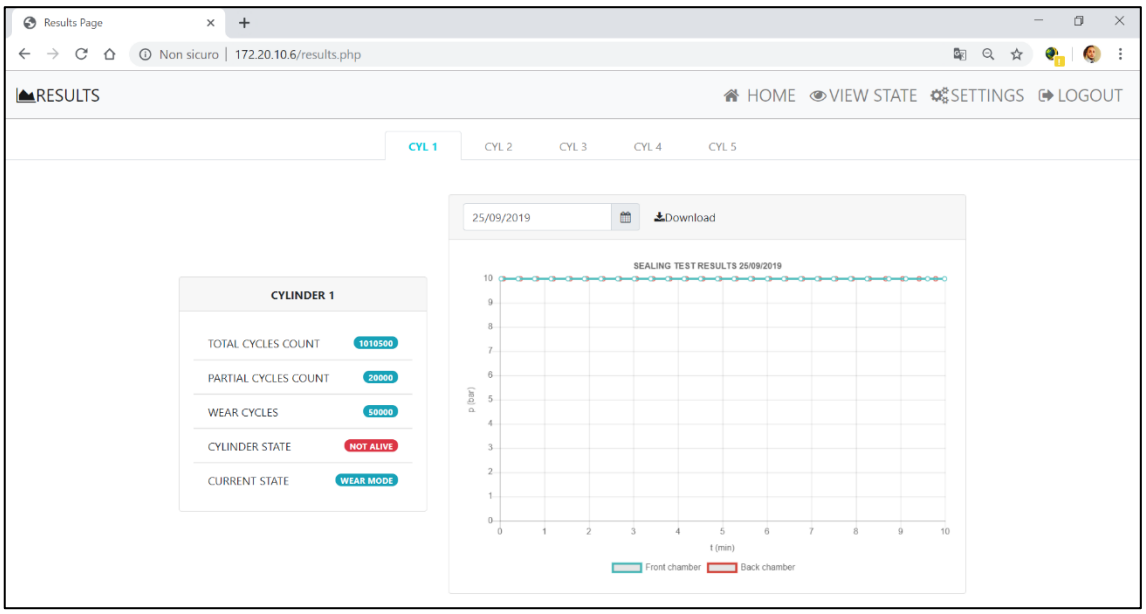

Figura 6-3: Results Page of the website

The PHP file containing this kind of page is named “results.php”, as it can be seen from the address bar. In the screenshot, it is possible to see a plot related to the results of the seal test performed on a particular date on the front chamber and the rear chamber of the first cylinder. Pressure is plotted on the y-axis of the plot, while time is represented on the x-axis. The values plotted are taken from the files stored inside the folders “/var/www/html/CYL1” and “/var/www/html/CYL2”, the files that contain the results of the seal tests, created during the tests as we have seen in previous paragraphs. It is also possible to download those files on the PC by pressing the “Download” button.

The screenshot shows that the site was prepared to host a higher number of cylinders and also to contain other features that we are not using at the moment and that have

not been connected to the rest of the test bench, for example information about the cylinders. This is a job left for future developments.

In order to get another example of results for the seal tests, we performed a seal test connecting a restrictor valve on the front chamber of the second cylinder, to have controlled artificial air losses from the front chamber on purpose. In the next figure we show the schematic for the pneumatic network in this case:

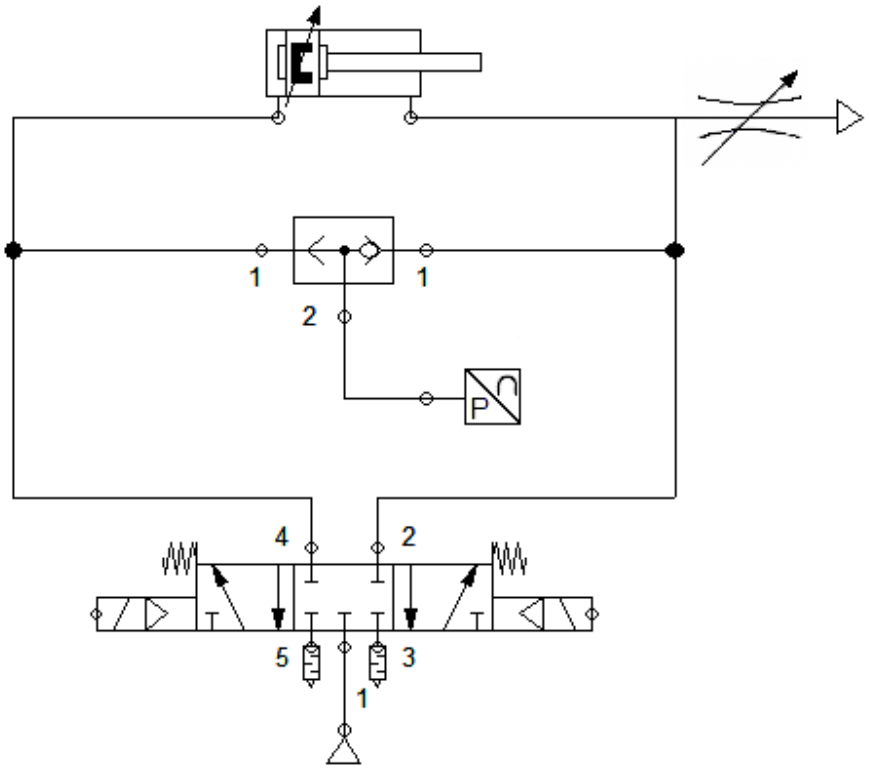

Figura 6-4: Pneumatic network with restrictor valve

The results obtained in this case are shown in the following figure:

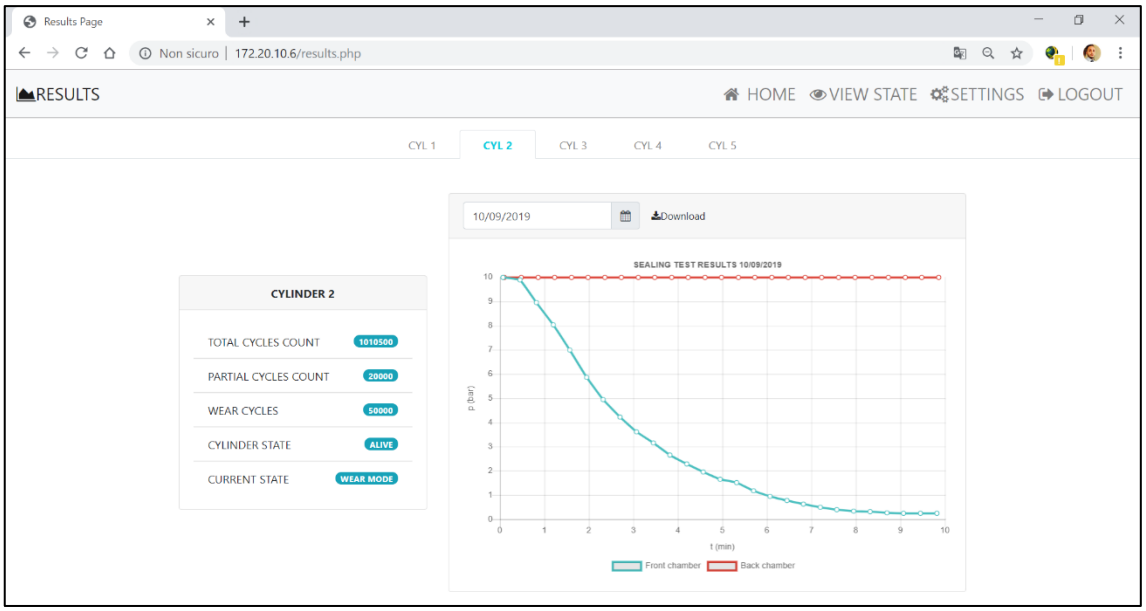

Figura 6-5: Results of the seal test with restrictor valve

Inside the plot it is possible to see two different lines: the red line for the back chamber, formed by stable pressure values approximately near to 10 bars, and the blue line for the front chamber, in which the pressure values progressively decrease due to the air losses on the front chamber.

7 CONCLUSIONS AND FUTURE DEVELOPMENTS

We succeeded in achieving the objectives set. The test bench is capable of complying with the required functionalities. The cylinders are controlled in such a way that the pistons can move in a reciprocating linear motion and also be stopped in order to perform some tests. Each cylinder can be handled separately or simultaneously, and this is due to both the hardware and software structure implemented.

The low cost Arduino board proved to be adequate for our purposes. Arduino can manage all the resources by means of cheap electronic veroboards, wires and components. Potential problems related to a lack of input/output pins are easily solved by means of small integrated circuits such as PCF8574N. Using interrupts, Arduino can provide short reaction times to any external event.

Moreover, the 10 bit resolution (2^10 = 1024 analog values) of the Arduino analog pins is enough for our task. Considering that the input values for the pressure range from 0 to 10 bars, the minimum step for the pressure is around 1/100 bar (10/1024), sufficient for the results we want to check during our seal tests.

The sampling frequencies that can be set through the Arduino timer interrupt are suitable with respect to what we need. We can just choose frequencies on the order of hertz, because we are just interested in seeing a general trend for the pressure of the air inside the chambers, so there is no need for a faster acquisition.

The code written in Arduino IDE appears organized in many distinct blocks, each one created with a specific functionality. Slight modifications result to be easily applicable thanks to the various parameters defined at the beginning of the code. By means of the structure named *cilindro* and the *cilindri* array, the operations that have to be performed on the pneumatic actuators are written in a compact and clear way. Each cylinder can be easily managed by means of for cycles, all at the same time or in an independent way according to what is needed.

As fas as future developments are concerned, it is natural to think about increasing the number of pneumatic actuators in the test bench up until to ten cylinders or even a bit more. The structure of the code is such that this task will not require any particular effort from future developers. It is just necessary to modify the N_CIL parameter at the beginning of the sketch, replicate some parts of the code and add other parts that could deal with the new electronic components in an entirely similar way to what has already been done. This type of expansion of the test bench simply requires some additional support circuitry realized in the exact same way of the current electronic network.

Future work could also involve an improvement of the Web Server hosted by the Raspberry Pi board. Arduino could send not only information about the pressure inside the actuators during the seal tests but also additional data such as total number of cycles completed by the actuators, partial number of cycles during a wear test, state of the cylinders etc., considering that some of these features have already been sketched inside the website. It is also possible to imagine a communication between the tools that works also the other way around. We could have a Web Server that not only is capable of collecting information but can also send commands to the Arduino board, allowing the user to interact with the system using a graphical interface. A user would be provided with the opportunity to start a test, stop the cylinders and many other operations through the website, meaning that he does not necessarily need to be in the same room of the test bench.

In this regard, developers could also be interested in making the website reachable even outside the network to which the Raspberry Pi is connected. For this purpose there are many possible solutions to allow the access to localhost from anywhere in the Web, for example secure tunnel services online such as “ngrok”, “PageKite”, “ProxyLocal” etc. or a router that offers the opportunity to the set the DMZ (demilitarized zone) with the private address of the Raspberry Pi.

APPENDIX A: ARDUINO CODE

Before the *setup()* block:

```
1.  //Working
2.
3.  //Reed: Analog 8 9 10 11 => 62 63 64 65 (porta K)
4.  //LED: Dal 2 all'11
5.  //Valvole: Dal 22 al 25
6.  //PCF: 53 52
7.  //Sensori pressione (analog) 54 55
8.
9.
10. #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
11. #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
12.
13. #include <Wire.h>
14.
15.
16. #define OUTPUT_COMPARE 0x1E84
17.                         //0x07A1 set to 31ms
18.                         //0x0F42 set to 62ms
19.                         //0x1E84 set to 125ms
20.                         //0x3D09 set to 250ms ---> 15625
21.
22.
23. #define N_CIL 2
24. #define USURA 0
25. #define TENUTA 1
26. #define IDLE_STATE 3
27. #define SWITCH 4
28. #define ZEROING_USURA 5
29. #define PRE_TENUTA 6
30. #define ZEROING_TENUTA 7
31. #define VALVE_OFF 8
32. #define PAUSA_USURA -1
33. #define PAUSA_TENUTA -2
34. #define ROTTO -3
35. #define OFF -4
36.
37. #define ANALOG_TRANSLATION 204
38. #define ANALOG_TO_BAR 0.0122
39. #define BROKEN_PRESSURE 0.5
40.
41.
42. #define PCF1 53
43. #define PCF2 52
44.
45. int i;
46. int j;
47. int butt=-1;
48.
49. unsigned long int TARGET_N = 20; //numero di corse
50. unsigned long int TARGET_T = 10000;
51. unsigned long int TARGET_PRE_T = 3000;
52. unsigned long int TARGET_VALVEOFF = 1000;
53.
54. typedef struct{
55.   int state;
56.
57.   bool TENUTA_ANT;
58.   long int N;
```

```
59.
60.   long int N_tot;
61.   long int T0;
62.   long int T0_blink;
63.   long int T0_pretenuta;
64.   long int T0_valveoff;
65.
66.   bool fineCorsa[2];
67.   int valvePin[2];
68.   int reed[2];
69.   int pressSens;
70.   bool valvePin_prestop[2];
71.
72.   int STOP_LED; //rosso
73.   int SEAL_LED; //giallo
74.   int WEAR_LED; //blu
75.   int SELECT_LED; //verde
76.   int BROKEN_LED;
77.   bool tenutaOK[2];
78.
79.
80.   } cilindro;
81.
82. cilindro cilindri[N_CIL];
83.
84. volatile byte PortBStatus;
85. volatile byte PortKStatus;
86. volatile byte PrevPortBStatus;
87. volatile byte PrevPortKStatus;
88.
89. bool PortKUpdated, PortBUpdated;
```

setup() block:

```
1.  void setup() {
2.
3.      Serial.begin(115200);
4.
5.      while(!Serial.available()){
6.
7.
8.      }
9.
10.     while(1){
11.       if(Serial.read()== '1') break;
12.       delay(5);
13.     }
14.
15.     timerInterruptSetup();
16.
17.   for(int n_cil=0;n_cil<N_CIL;n_cil++){
18.     cilindri[n_cil].state=IDLE_STATE;
19.
20.     cilindri[n_cil].N=0;
21.     cilindri[n_cil].N_tot=0;
22.     cilindri[n_cil].tenutaOK[0]=true;
23.     cilindri[n_cil].tenutaOK[1]=true;
24.
25.   }
26.
27.  for(int n_cil=0; n_cil<N_CIL; n_cil++){
```

```
28.
29.    cilindri[n_cil].pressSens=n_cil+54;
30.    pinMode(cilindri[n_cil].pressSens, INPUT);
31.
32.     for(i=0;i<2;i++){
33.
34.       cilindri[n_cil].valvePin[i]=n_cil*2+i+22;
35.       pinMode(cilindri[n_cil].valvePin[i], OUTPUT);
36.       digitalWrite(cilindri[n_cil].valvePin[i], LOW);
37.       cilindri[n_cil].valvePin_prestop[i]=0;
38.     }
39.
40.     for(i=0;i<2;i++){
41.       cilindri[n_cil].reed[i]=n_cil*2+i+62;
42.       pinMode(cilindri[n_cil].reed[i], INPUT);
43.       cilindri[n_cil].fineCorsa[i]=digitalRead(cilindri[n_cil].reed[i]);
44.     }
45.
46.
47.     cilindri[n_cil].STOP_LED=n_cil+2;
48.     pinMode(cilindri[n_cil].STOP_LED, OUTPUT);
49.     digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
50.     cilindri[n_cil].SEAL_LED=n_cil+4;
51.     pinMode(cilindri[n_cil].SEAL_LED, OUTPUT);
52.     cilindri[n_cil].WEAR_LED=n_cil+6;
53.     pinMode(cilindri[n_cil].WEAR_LED, OUTPUT);
54.     cilindri[n_cil].SELECT_LED=n_cil+8;
55.     pinMode(cilindri[n_cil].SELECT_LED, OUTPUT);
56.     digitalWrite(cilindri[n_cil].SELECT_LED, HIGH);
57.     cilindri[n_cil].BROKEN_LED=11-n_cil;
58.     pinMode(cilindri[n_cil].BROKEN_LED, OUTPUT);
59.
60.   }
61.
62.
63.   Wire.begin();
64.     delay(500);
65.
66.
67.     Serial.println(expanderRead(0x40>>1), BIN);
68.     Serial.println(expanderRead(0x42>>1), BIN);
69.
70. //PREPARE PIN CHANGE INTERRUPT INTERRUPTS
71.
72.   //PORTB
73.   sbi(PCICR, PCIE0);
74.   sbi(PCMSK0, PCINT0); //DIGITAL PIN 53
75.   sbi(PCMSK0, PCINT1); //DIGITAL PIN 52
76.
77.   //PORTD
78.   sbi(PCICR, PCIE2);
79.   sbi(PCMSK2, PCINT18); //DIGITAL PIN 62
80.   sbi(PCMSK2, PCINT19); //DIGITAL PIN 63
81.   sbi(PCMSK2, PCINT16); //DIGITAL PIN 64
82.   sbi(PCMSK2, PCINT17); //DIGITAL PIN 65
83.
84.   PortBUpdated=false;
85.   PortKUpdated=false;
86.
87.
88.
89. PrevPortKStatus=PINK&0x0F;
90. Serial.println(PrevPortKStatus, BIN);
91. }
```

loop() block:

```
void loop() {


  if((PortBUpdated || PortKUpdated ))
  {
    if(PortBUpdated){

      PortBUpdated=false;
      butt=GetSwitchPress('B');
    }

    else if(PortKUpdated){

        byte temp=PortKStatus;
        PortKStatus=(PrevPortKStatus^PortKStatus)&PortKStatus;
        PrevPortKStatus=temp;

        PortKUpdated=false;

         byte checkbit=1;
         for(int n_cil=0;n_cil<N_CIL; n_cil++){
          for(int i=0; i<2;i++){

              if((PortKStatus&checkbit)>0){
                cilindri[n_cil].fineCorsa[i]=true;

              }
              else{
                cilindri[n_cil].fineCorsa[i]=false;

              }
              checkbit=checkbit*2;
        }
    }

    }
  }

  for(i=0; i<N_CIL; i++)
  {check_state(i);}

  for(j=0; j<N_CIL; j++)
  {update_cil(j);}

  update_butt();

  butt=-1;
}
```

After the *loop()* block:

“check_state” function:

```
void check_state(int n_cil){

    if(cilindri[n_cil].state==ZEROING_USURA){
```

```
4.          digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
5.          digitalWrite(cilindri[n_cil].valvePin[0], LOW);
6.          blinkWearLed(n_cil, 100);
7.
8.          if(digitalRead(cilindri[n_cil].reed[0])){
9.
10.           cilindri[n_cil].fineCorsa[0]=digitalRead(cilindri[n_cil].reed[0]);
11.           cilindri[n_cil].fineCorsa[1]=digitalRead(cilindri[n_cil].reed[1]);
12.
13.           digitalWrite(cilindri[n_cil].WEAR_LED, HIGH);
14.
15.           cilindri[n_cil].state=USURA;
16.
17.         }
18.
19.       }
20.
21.     if(cilindri[n_cil].state == USURA){
22.
23.         if(cilindri[n_cil].N == TARGET_N){
24.           cilindri[n_cil].N_tot += cilindri[n_cil].N ;
25.           cilindri[n_cil].N = 0;
26.           digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
27.           cilindri[n_cil].state = ZEROING_TENUTA;
28.           cilindri[n_cil].T0_blink=millis();
29.
30.          }
31.       }
32.
33.     if(cilindri[n_cil].state==ZEROING_TENUTA){
34.         digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
35.         digitalWrite(cilindri[n_cil].valvePin[0], LOW);
36.
37.           if(digitalRead(cilindri[n_cil].reed[0])){
38.
39.           cilindri[n_cil].fineCorsa[0]=digitalRead(cilindri[n_cil].reed[0]);
40.           cilindri[n_cil].fineCorsa[1]=digitalRead(cilindri[n_cil].reed[1]);
41.
42.           cilindri[n_cil].state=VALVE_OFF;
43.
44.           cilindri[n_cil].TENUTA_ANT=true;
45.           digitalWrite(cilindri[n_cil].valvePin[0], LOW);
46.
47.           digitalWrite(cilindri[n_cil].valvePin[1], LOW);
48.
49.           cilindri[n_cil].T0_valveoff=millis();
50.           cilindri[n_cil].T0_blink=millis();
51.
52.         }
53.         blinkSealLed(n_cil, 100);
54.       }
55.
56.     if(cilindri[n_cil].state==VALVE_OFF){
57.       if(millis()-cilindri[n_cil].T0_valveoff>TARGET_VALVEOFF){
58.        if(cilindri[n_cil].TENUTA_ANT){
59.         digitalWrite(cilindri[n_cil].valvePin[1], HIGH);
60.         Serial.println("s" + String(n_cil));
61.
62.        }
63.        else{
64.         digitalWrite(cilindri[n_cil].valvePin[0], HIGH);
65.
```

```
66.         }
67.         cilindri[n_cil].T0_pretenuta=millis();
68.         cilindri[n_cil].T0_blink=millis();
69.
70.         cilindri[n_cil].state=PRE_TENUTA;
71.
72.        }
73.        blinkSealLed(n_cil, 200);
74.       }
75.
76.      if(cilindri[n_cil].state==PRE_TENUTA){
77.          if(millis()-cilindri[n_cil].T0_pretenuta>TARGET_PRE_T){
78.            cilindri[n_cil].T0=millis();
79.            cilindri[n_cil].T0_blink=millis();
80.            cilindri[n_cil].state=TENUTA;
81.
82.            if(cilindri[n_cil].TENUTA_ANT){
83.              digitalWrite(cilindri[n_cil].valvePin[1], LOW);
84.
85.            }
86.            else{
87.              digitalWrite(cilindri[n_cil].valvePin[0], LOW);
88.
89.             }
90.          }
91.           blinkSealLed(n_cil, 200);
92.        }
93.
94.      if(cilindri[n_cil].state == TENUTA){
95.          if(cilindri[n_cil].TENUTA_ANT){
96.            if((millis()-cilindri[n_cil].T0)>TARGET_T){
97.              float p =(float)(analogRead(cilindri[n_cil].pressSens)-
    ANALOG_TRANSLATION)*ANALOG_TO_BAR;
98.              if(p < BROKEN_PRESSURE){
99.
100.                    cilindri[n_cil].tenutaOK[0]=false;
101.
102.                 }
103.
104.                  cilindri[n_cil].state=SWITCH;
105.
106.                  digitalWrite(cilindri[n_cil].valvePin[0], HIGH);
107.
108.                  cilindri[n_cil].TENUTA_ANT=false;
109.                  cilindri[n_cil].T0_blink=millis();
110.
111.                }
112.              }
113.              else{
114.                if((millis()-cilindri[n_cil].T0)>TARGET_T){
115.                  float p =(float)(analogRead(cilindri[n_cil].pressSens)-
    ANALOG_TRANSLATION)*ANALOG_TO_BAR;
116.
117.                  if(p < BROKEN_PRESSURE){
118.
119.                    cilindri[n_cil].tenutaOK[1]=false;
120.
121.                  }
122.                  if(cilindri[n_cil].tenutaOK[0]==true && cilindri[n_cil].tenu
    taOK[1]==true)  {
123.                   digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
124.                   cilindri[n_cil].state = ZEROING_USURA;
125.
126.                  }
127.                  else{
128.                    digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
```

```
                    digitalWrite(cilindri[n_cil].SELECT_LED, LOW);
                    digitalWrite(cilindri[n_cil].BROKEN_LED, HIGH);
                    cilindri[n_cil].state = ROTTO;
                  }
                }

              }
              blinkSealLed(n_cil, 500);
            }

          if(cilindri[n_cil].state==SWITCH){
              if(cilindri[n_cil].fineCorsa[1]){
                 cilindri[n_cil].state=VALVE_OFF;

                 digitalWrite(cilindri[n_cil].valvePin[0], LOW);

                 digitalWrite(cilindri[n_cil].valvePin[1], LOW);

                 cilindri[n_cil].T0_valveoff=millis();
                 cilindri[n_cil].T0_blink=millis();

              }
              blinkSealLed(n_cil, 100);
            }

          if(cilindri[n_cil].state == PAUSA_USURA){
            blinkStopLed(n_cil, 100);

            }

            if(cilindri[n_cil].state == PAUSA_TENUTA){
            blinkStopLed(n_cil, 500);

            }
      }
```

“update_cil” function:

```
void update_cil(int n_cil){

  if(cilindri[n_cil].state == USURA && cilindri[n_cil].fineCorsa[0]){

        cilindri[n_cil].fineCorsa[0]=0;

        if(cilindri[n_cil].N <TARGET_N-1){

          digitalWrite(cilindri[n_cil].valvePin[0],HIGH);
          digitalWrite(cilindri[n_cil].valvePin[1],LOW);

        };

        cilindri[n_cil].N++;

  }

  if(cilindri[n_cil].state == USURA && cilindri[n_cil].fineCorsa[1]){

        cilindri[n_cil].fineCorsa[1]=0;

        if(cilindri[n_cil].N <TARGET_N){

```

```
            digitalWrite(cilindri[n_cil].valvePin[0],LOW);
            digitalWrite(cilindri[n_cil].valvePin[1],HIGH);

          };

          cilindri[n_cil].N++;

    }

}
```

“update_butt” function:

```
void update_butt(){

    if(butt==10000) //ON OFF TOTALE
      {
        for(int n_cil=0; n_cil<N_CIL; n_cil++){
          if(cilindri[n_cil].state==OFF){
            cilindri[n_cil].state = IDLE_STATE;
            digitalWrite(cilindri[n_cil].SELECT_LED, HIGH);
            digitalWrite(cilindri[n_cil].STOP_LED, HIGH);

            }
          else{
            cilindri[n_cil].N_tot += cilindri[n_cil].N ;
            cilindri[n_cil].N = 0;
            cilindri[n_cil].state = OFF;
            digitalWrite(cilindri[n_cil].STOP_LED, LOW);
            digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
            digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
            digitalWrite(cilindri[n_cil].SELECT_LED, LOW);
            digitalWrite(cilindri[n_cil].BROKEN_LED, LOW);

            }
        }
      }

      if(butt==11000) //ON OFF UN CILINDRO
      {
          if(cilindri[0].state==OFF){
            cilindri[0].state = IDLE_STATE;
            digitalWrite(cilindri[0].SELECT_LED, HIGH);
            digitalWrite(cilindri[0].STOP_LED, HIGH);

            }
          else{
            cilindri[0].N_tot += cilindri[0].N ;
            cilindri[0].N = 0;
            cilindri[0].state = OFF;
            digitalWrite(cilindri[0].STOP_LED, LOW);
            digitalWrite(cilindri[0].SEAL_LED, LOW);
            digitalWrite(cilindri[0].WEAR_LED, LOW);
            digitalWrite(cilindri[0].SELECT_LED, LOW);

            }

      }

       if(butt==12000) //ON OFF UN CILINDRO
      {
```

```
49.           if(cilindri[1].state==OFF){
50.             cilindri[1].state = IDLE_STATE;
51.             digitalWrite(cilindri[1].SELECT_LED, HIGH);
52.             digitalWrite(cilindri[1].STOP_LED, HIGH);
53.
54.             }
55.           else{
56.             cilindri[1].N_tot += cilindri[1].N ;
57.             cilindri[1].N = 0;
58.             cilindri[1].state = OFF;
59.             digitalWrite(cilindri[1].STOP_LED, LOW);
60.             digitalWrite(cilindri[1].SEAL_LED, LOW);
61.             digitalWrite(cilindri[1].WEAR_LED, LOW);
62.             digitalWrite(cilindri[1].SELECT_LED, LOW);
63.
64.             }
65.
66.       }
67.
68.     if(butt==300) //PAUSA STOP TOTALE
69.      {
70.        for(int n_cil=0; n_cil<N_CIL; n_cil++){
71.
72.             if(cilindri[n_cil].state==USURA){
73.               cilindri[n_cil].state = PAUSA_USURA;
74.               cilindri[n_cil].T0_blink=millis();
75.               digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
76.
77.               cilindri[n_cil].valvePin_prestop[0]=digitalRead(cilindri[n_cil].
    valvePin[0]);
78.               cilindri[n_cil].valvePin_prestop[1]=digitalRead(cilindri[n_cil].
    valvePin[1]);
79.
80.               digitalWrite(cilindri[n_cil].valvePin[0], LOW);
81.               digitalWrite(cilindri[n_cil].valvePin[1], LOW);
82.              }
83.             else if(cilindri[n_cil].state==PRE_TENUTA || cilindri[n_cil].state
    ==TENUTA || cilindri[n_cil].state==SWITCH ){
84.
85.               cilindri[n_cil].state = PAUSA_TENUTA;
86.               cilindri[n_cil].T0_blink=millis();
87.               digitalWrite(cilindri[n_cil].SEAL_LED, LOW);
88.
89.               digitalWrite(cilindri[n_cil].valvePin[0], LOW);
90.               digitalWrite(cilindri[n_cil].valvePin[1], LOW);
91.
92.             }
93.              else if(cilindri[n_cil].state == PAUSA_USURA){
94.               cilindri[n_cil].state = IDLE_STATE;
95.               digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
96.
97.               digitalWrite(cilindri[n_cil].valvePin[0], LOW);
98.               digitalWrite(cilindri[n_cil].valvePin[1], LOW);
99.
100.                    cilindri[n_cil].N_tot += cilindri[n_cil].N;
101.                    cilindri[n_cil].N = 0;
102.
103.                  }
104.                  else if(cilindri[n_cil].state == PAUSA_TENUTA){
105.                    cilindri[n_cil].state = IDLE_STATE;
106.                    digitalWrite(cilindri[n_cil].STOP_LED, HIGH);
107.
108.                   digitalWrite(cilindri[n_cil].valvePin[0], LOW);
109.                   digitalWrite(cilindri[n_cil].valvePin[1], LOW);
110.
111.                    }
```

```
112.
113.                 }
114.              }
115.
116.            if(butt==310) //PAUSA STOP UN CILINDRO
117.             {
118.
119.
120.                    if(cilindri[0].state==USURA){
121.                      cilindri[0].state = PAUSA_USURA;
122.                      cilindri[0].T0_blink=millis();
123.                      digitalWrite(cilindri[0].WEAR_LED, LOW);
124.
125.                      cilindri[0].valvePin_prestop[0]=digitalRead(cilindri[0].v
    alvePin[0]);
126.                      cilindri[0].valvePin_prestop[1]=digitalRead(cilindri[0].v
    alvePin[1]);
127.
128.                      digitalWrite(cilindri[0].valvePin[0], LOW);
129.                      digitalWrite(cilindri[0].valvePin[1], LOW);
130.                     }
131.                    else if(cilindri[0].state==PRE_TENUTA || cilindri[0].state=
    =TENUTA || cilindri[0].state==SWITCH ){
132.
133.                      cilindri[0].state = PAUSA_TENUTA;
134.                      cilindri[0].T0_blink=millis();
135.                      digitalWrite(cilindri[0].SEAL_LED, LOW);
136.
137.                      digitalWrite(cilindri[0].valvePin[0], LOW);
138.                      digitalWrite(cilindri[0].valvePin[1], LOW);
139.
140.                    }
141.                     else if(cilindri[0].state == PAUSA_USURA){
142.                      cilindri[0].state = IDLE_STATE;
143.                      digitalWrite(cilindri[0].STOP_LED, HIGH);
144.
145.                      digitalWrite(cilindri[0].valvePin[0], LOW);
146.                      digitalWrite(cilindri[0].valvePin[1], LOW);
147.
148.                      cilindri[0].N_tot += cilindri[0].N;
149.                      cilindri[0].N = 0;
150.
151.                     }
152.                     else if(cilindri[0].state == PAUSA_TENUTA){
153.                       cilindri[0].state = IDLE_STATE;
154.                       digitalWrite(cilindri[0].STOP_LED, HIGH);
155.
156.                      digitalWrite(cilindri[0].valvePin[0], LOW);
157.                      digitalWrite(cilindri[0].valvePin[1], LOW);
158.
159.                       }
160.
161.
162.              }
163.
164.        if(butt==320) //PAUSA STOP UN CILINDRO
165.             {
166.
167.
168.                    if(cilindri[1].state==USURA){
169.                      cilindri[1].state = PAUSA_USURA;
170.                      cilindri[1].T0_blink=millis();
171.                      digitalWrite(cilindri[1].WEAR_LED, LOW);
172.
173.                      cilindri[1].valvePin_prestop[0]=digitalRead(cilindri[1].v
    alvePin[0]);
```

```
                    cilindri[1].valvePin_prestop[1]=digitalRead(cilindri[1].valvePin[1]);

                    digitalWrite(cilindri[1].valvePin[0], LOW);
                    digitalWrite(cilindri[1].valvePin[1], LOW);
                   }
                  else if(cilindri[1].state==PRE_TENUTA || cilindri[1].state==TENUTA || cilindri[1].state==SWITCH ){

                    cilindri[1].state = PAUSA_TENUTA;
                    cilindri[1].T0_blink=millis();
                    digitalWrite(cilindri[1].SEAL_LED, LOW);

                    digitalWrite(cilindri[1].valvePin[0], LOW);
                    digitalWrite(cilindri[1].valvePin[1], LOW);

                  }
                   else if(cilindri[1].state == PAUSA_USURA){
                    cilindri[1].state = IDLE_STATE;
                    digitalWrite(cilindri[1].STOP_LED, HIGH);

                    digitalWrite(cilindri[1].valvePin[0], LOW);
                    digitalWrite(cilindri[1].valvePin[1], LOW);

                    cilindri[1].N_tot += cilindri[1].N;
                    cilindri[1].N = 0;

                   }
                   else if(cilindri[1].state == PAUSA_TENUTA){
                     cilindri[1].state = IDLE_STATE;
                     digitalWrite(cilindri[1].STOP_LED, HIGH);

                    digitalWrite(cilindri[1].valvePin[0], LOW);
                    digitalWrite(cilindri[1].valvePin[1], LOW);

                     }


            }

          if (butt==200)   //USURA TOTALE
            {
              for(int n_cil=0; n_cil<N_CIL; n_cil++){
                if(cilindri[n_cil].state == PAUSA_USURA){
                  cilindri[n_cil].state = USURA;
                  digitalWrite(cilindri[n_cil].STOP_LED, LOW);

                     digitalWrite(cilindri[n_cil].valvePin[0], cilindri[n_cil].valvePin_prestop[0]);
                     digitalWrite(cilindri[n_cil].valvePin[1], cilindri[n_cil].valvePin_prestop[1]);

                     digitalWrite(cilindri[n_cil].WEAR_LED, HIGH);
                 }

                else if(cilindri[n_cil].state == IDLE_STATE){

                     cilindri[n_cil].state = ZEROING_USURA;

                     digitalWrite(cilindri[n_cil].STOP_LED, LOW);
                 }
              }
            }

      if (butt==210)   //USURA UN CILINDRO
            {
```

```
236.
237.                 if(cilindri[0].state == PAUSA_USURA){
238.                   cilindri[0].state = USURA;
239.                   digitalWrite(cilindri[0].STOP_LED, LOW);
240.
241.                       digitalWrite(cilindri[0].valvePin[0], cilindri[0].valve
    Pin_prestop[0]);
242.                       digitalWrite(cilindri[0].valvePin[1], cilindri[0].valve
    Pin_prestop[1]);
243.
244.                       digitalWrite(cilindri[0].WEAR_LED, HIGH);
245.                  }
246.
247.                 else if(cilindri[0].state == IDLE_STATE){
248.
249.                       cilindri[0].state = ZEROING_USURA;
250.
251.                       digitalWrite(cilindri[0].STOP_LED, LOW);
252.                  }
253.
254.             }
255.
256.       if (butt==220)   //USURA UN CILINDRO
257.             {
258.
259.                 if(cilindri[1].state == PAUSA_USURA){
260.                   cilindri[1].state = USURA;
261.                   digitalWrite(cilindri[1].STOP_LED, LOW);
262.
263.                       digitalWrite(cilindri[1].valvePin[0], cilindri[1].valve
    Pin_prestop[0]);
264.                       digitalWrite(cilindri[1].valvePin[1], cilindri[1].valve
    Pin_prestop[1]);
265.
266.                       digitalWrite(cilindri[1].WEAR_LED, HIGH);
267.                  }
268.
269.                 else if(cilindri[1].state == IDLE_STATE){
270.
271.                       cilindri[1].state = ZEROING_USURA;
272.
273.                       digitalWrite(cilindri[1].STOP_LED, LOW);
274.                  }
275.
276.             }
277.
278.           if(butt==100)  //TENUTA TOTALE
279.           {
280.             for(int n_cil=0; n_cil<N_CIL; n_cil++){
281.               if(cilindri[n_cil].state==USURA || cilindri[n_cil].state == IDL
    E_STATE){
282.                 digitalWrite(cilindri[n_cil].WEAR_LED, LOW);
283.                 digitalWrite(cilindri[n_cil].STOP_LED, LOW);
284.                 cilindri[n_cil].state = ZEROING_TENUTA;
285.
286.
287.             }
288.             else if(cilindri[n_cil].state == PAUSA_TENUTA){
289.                 cilindri[n_cil].state = ZEROING_TENUTA;
290.                 digitalWrite(cilindri[n_cil].STOP_LED, LOW);
291.
292.               }
293.           }
294.       }
295.
296.       if(butt==110)  //TENUTA UN CILINDRO
```

```
297.           {
298.
299.               if(cilindri[0].state==USURA || cilindri[0].state == IDLE_STATE)
    {
300.                 digitalWrite(cilindri[0].WEAR_LED, LOW);
301.                 digitalWrite(cilindri[0].STOP_LED, LOW);
302.                 cilindri[0].state = ZEROING_TENUTA;
303.
304.
305.             }
306.             else if(cilindri[0].state == PAUSA_TENUTA){
307.                 cilindri[0].state = ZEROING_TENUTA;
308.                 digitalWrite(cilindri[0].STOP_LED, LOW);
309.
310.               }
311.
312.       }
313.
314.       if(butt==120)  //TENUTA UN CILINDRO
315.           {
316.
317.               if(cilindri[1].state==USURA || cilindri[1].state == IDLE_STATE)
    {
318.                 digitalWrite(cilindri[1].WEAR_LED, LOW);
319.                 digitalWrite(cilindri[1].STOP_LED, LOW);
320.                 cilindri[1].state = ZEROING_TENUTA;
321.
322.
323.             }
324.             else if(cilindri[1].state == PAUSA_TENUTA){
325.                 cilindri[1].state = ZEROING_TENUTA;
326.                 digitalWrite(cilindri[1].STOP_LED, LOW);
327.
328.               }
329.
330.        }
331.       }
```

Blinking functions:

```
1.  void blinkSealLed(int n_cil, int T){
2.    if(millis()-cilindri[n_cil].T0_blink>T){
3.      digitalWrite(cilindri[n_cil].SEAL_LED, !digitalRead(cilindri[n_cil].SEAL_L
    ED));
4.      cilindri[n_cil].T0_blink=millis();
5.    }
6.  }
7.
8.
9.  void blinkWearLed(int n_cil, int T){
10.   if(millis()-cilindri[n_cil].T0_blink>T){
11.     digitalWrite(cilindri[n_cil].WEAR_LED, !digitalRead(cilindri[n_cil].WEAR_L
    ED));
12.     cilindri[n_cil].T0_blink=millis();
13.   }
14. }
15.
16. void blinkStopLed(int n_cil, int T){
17.   if(millis()-cilindri[n_cil].T0_blink>T){
18.     digitalWrite(cilindri[n_cil].STOP_LED, !digitalRead(cilindri[n_cil].STOP_L
    ED));
```

```
    cilindri[n_cil].T0_blink=millis();
  }
}
```

“GetSwitchPress” function:

```
int GetSwitchPress(char Port){
f(Port=='K')
 {
   byte SwitchState=PortKStatus;
   switch(SwitchState){

     case 1:
       return  0;
     case 2:
       return  1;
     case 4:
       return  2;
     case 8:
       return  3;
     case 5:
       return 5;
     case 6:
       return 6;
     case 9:
       return 9;
     case 10:
       return 10;

     default:
       return -1;
   }
 }
else if(Port=='B'){
   if(PortBStatus==1){ //53

        byte data = expanderRead(0x40>>1);
        data = ~data;
        byte SwitchState = data;
        switch(SwitchState){

          case 1:
          return 12000;
          case 2:
          return 220;
          case 4:
          return 11000;
          case 8:
          return 210;
          case 16:
          return 310;
          case 32:
          return 110;
          case 64:
          return 320;
          case 128:
          return 120;
        default:
          return -1;
        }
      }
```

```
      if(PortBStatus==2){ //52

        byte data = expanderRead(0x42>>1);
        data = ~data;
        byte SwitchState = data;
        switch(SwitchState){

          case 1:
          return 10000;
          case 2:
          return 200;
          case 64:
          return 300;
          case 128:
          return 100;

        default:
          return -1;
        }

      }
    }
}
```

Interrupt Service Routines:

```
ISR(PCINT0_vect)
{
  PortBStatus=PINB;
  PortBStatus=~PortBStatus;

  PortBStatus&=0x3;

  if(PortBStatus==1 ||PortBStatus==2){

    PortBUpdated=true;
  }
}

ISR(PCINT2_vect)
{
  PortKStatus=PINK;
  PortKStatus&=0x0F;

  if(PortKStatus>0){

    PortKUpdated=true;

  }
}

ISR(TIMER1_COMPA_vect){

for(int n_cil=0; n_cil<N_CIL; n_cil++){
     if(cilindri[n_cil].state==TENUTA){

        float p =(float)(analogRead(cilindri[n_cil].pressSens)-
ANALOG_TRANSLATION)*ANALOG_TO_BAR;

        String X = String(n_cil);
        X= X + X;
```

```
35.
36.         X = X + "," + String(((float)(millis()-
    cilindri[n_cil].T0)/1000)) + "," + String(p);
37.         Serial.println(X);
38.
39.
40.        }
41.    }
42. }
```

Functions for the PCF8574N:

```
1.  void expanderWrite(byte _data, byte address ) {
2.    Wire.beginTransmission(address); //es address = 0x42>>1
3.    Wire.write(_data);
4.    Wire.endTransmission();
5.  }
6.
7.  byte expanderRead(byte address) {
8.    byte _data;
9.    Wire.requestFrom(address, 1);
10.   if(Wire.available()) {
11.     _data = Wire.read();
12.   }
13.   return _data;
14. }
```

“timerInterruptSetup” function:

```
1.  void timerInterruptSetup(){
2.      cli();  //disable the global interrupt
3.      //Timer/Counter 1
4.      TCCR1A = 0x00;
5.      //TCCR1B = (_BV(WGM12)) | (_BV(CS11)) | (_BV(CS10));  //CTC mode, clk/64
6.      TCCR1B = (_BV(WGM12)) | (_BV(CS12)); //CTC mode, Set prescaler to 256
7.      OCR1A = OUTPUT_COMPARE; //set timer frequency
8.      TCNT1 = 0x00; //initialize the counter
9.      TIMSK1 = _BV(OCIE1A);  //Output Compare Match Interrupt Enable
10.     sei(); //enable global interrupt
11. }
```

APPENDIX B: RASPBERRY PI CODE

```
#!/usr/bin/python2.7

import serial
import time
import csv
import os
import datetime

ser=serial.Serial('/dev/ttyACM0', 115200)
time.sleep(2)
ser.flushInput()
ser.write(b'1')

while True:
    while ser.inWaiting()==0:
        pass
    data= ser.readline()
    print(data)
    strindata=str(data)

    if strindata.startswith("00,"):
        csvwriter0.write(strindata[3:])
        #csvwriter0.write('\n')
        csvwriter0.flush()
        os.fsync(csvwriter0.fileno())
    elif strindata.startswith("11,"):
        csvwriter1.write(strindata[3:])
        #csvwriter1.write('\n')
        csvwriter1.flush()
        os.fsync(csvwriter1.fileno())
    elif strindata.startswith("s0"):
        #s="/var/www/html/CYL1/"+datetime.datetime.now().strftime("%Y_%m_%d-%H.%M")+".txt"
        s="/var/www/html/CYL1/"+datetime.datetime.now().strftime("%Y-%m-%d")+".csv"
        csvwriter0=open(s,"wb")
    elif strindata.startswith("s1"):
        s="/var/www/html/CYL2/"+datetime.datetime.now().strftime("%Y-%m-%d")+".csv"
        csvwriter1=open(s,"wb")

```

References

[1] "Slideshare," [Online]. Available: https://www.slideshare.net/waqar310/pneumatic-valves. [Accessed 14th June 2019].

[2] "Slideshare," [Online]. Available: https://www.slideshare.net/waqar310/switches-and-sensors. [Accessed 19th April 2019].

[3] "HACKADAY," [Online]. Available: https://hackaday.com/2015/12/09/embed-with-elliot-debounce-your-noisy-buttons-part-i/. [Accessed 22nd March 2019].

[4] "Texas Instruments," [Online]. Available: http://www.ti.com/lit/ds/symlink/pcf8574.pdf. [Accessed 28th April 2019].

[5] "Elettronica per passione," [Online]. Available: https://www.vincenzov.net/tutorial/motoridc/driver.htm. [Accessed 20th April 2019].

[6] "DanieleAlberti," [Online]. Available: https://www.danielealberti.it/2016/05/i-mosfet-e-arduino-come-pilotare-carichi.html. [Accessed 24th April 2019].

[7] "Sparkfun," [Online]. Available: https://www.sparkfun.com/datasheets/Components/Buttons/AN104.pdf. [Accessed 8th May 2019].

[8] "CircuitMess," [Online]. Available: https://www.circuitmess.com/gpio-expander-guide/. [Accessed 3rd June 2019].

[9] "RaspberryPi," [Online]. Available: https://www.raspberrypi.org/documentation/faqs/. [Accessed 6th September 2019].

[10] "Programiz," [Online]. Available: https://www.programiz.com/python-programming/datetime/strftime. [Accessed 05th July 2019].

[11] "Selfserver," [Online]. Available: http://www.selfserver.it/glossario/web-server. [Accessed 07th July 2019].