POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Automation and revamping of a lathe machine with Arduino control boards

Tutors:

Prof. Luigi Mazza

Prof. Andrea Mura

Candidate:

Giorgio Gorgerino

Academic year 2023/2024

Abstract of the thesis

Mechanical operations are performed by several machines depending on the type of product to be realized, both at industrial and laboratory levels. These machines can be both manual actuated or operate automatically and the lathe machine falls inside this category, which is used to work on revolutionary pieces.

This thesis work concerns with the description and analysis of all the features implemented aiming to automate a lathe machine: starting from a manual instrument, the goal of the work is to modify it in order to obtain a low-cost automatic machine able to perform lathing operations in a didactic laboratory environment.

The project can be subdivided into 3 main goals. Firstly, the implementation of an Arduino code useful to guide the movement of the working tool, through the command of suitable drivers, motors and mechanical elements able to convert the software program into mechanical actions.

Three different types of processes have been implemented depending on the shape of the final product to obtain: with a human-machine interface, the user can choose the desired work cycle and set all the parameters needed. In the set of all the obtainable shapes, the processes developed regard the cylindrical and the conical turning.

Furthermore, the more electrical part has been developed, concerning the design, simulation and practical realization of an electrical circuit able to drive the permanent magnet direct current motor useful to generate the rotational movement of the piece under work. Specifically, the electrical circuit built is a PWM network, which is able to control the voltage, and the current sent to the motor: the tuning of these physical quantities is aimed to obtain different rotational speeds, required by the user needs. To test the topology of the electrical circuit, a smaller one has been realized, handling a lower electrical power than the full power sustained by the motor itself. Other solutions are proposed to generate and handle a greater power for the motor.

Finally, the link between the previous two parts have been faced: the implementation of the Arduino code useful to drive the DC motor and control its rotational speed and the final communication between the two Arduino boards in order to perform a desired working cycle.

The last steps of the project have been the mechanical design with a 3D CAD software and realization of the additional components needed to complete the machine and the final working tests to check the correctness of the codes previously written. For a more complete test procedure, different working conditions have been implemented firstly at a code-level and then realized in the real world.

Table of Contents

List of Figures

List of Codes

List of tables

1. Introduction

Mechanical operations in production sites are performed by several types of machines, depending on the features and dimensions of the products to obtain. One of the most common machines is the lathe which is used to perform turning operations for cylindrical or conical shapes. Starting from a cylindrical raw piece in terms of dimensions and roughness level, the turning process outputs various shapes around its rotation axis, realized by a moving cutting tool.

The circular cutting motion is entrusted to the piece, mounted on the spindle, while the rectilinear feed motion is entrusted to the tool, mounted on the carriages.

As the majority of the mechanical machines, nowadays there exist both manual and automatic machines: the goal of this thesis is to preliminarily automatize a manual parallel lathe, able to perform some predetermined programs of working cycles on pieces in a limited range of dimensions and according to the parameters inserted by the operator.

The lathe considered is a manual machine and used for didactic purposes in a laboratory, due to its restricted dimensions of the products that can be processed. Specifically, the instrument is a SOGI M1-100 manual lathe (Fig. 1).

Figure 1: SOGI M1-100 manual lathe

To accomplish the various industrial needs, the sizes of the mechanical machines can differ greatly, in terms of physical dimensions and power involved during the operations, but the key elements that can be found on these are almost the same [01]:

- **Headstock**: located on the left side of the machine, it houses the main spindle, the motor that drives the spindle with the associated mechanical transmission and the user commands. The primary function of the headstock is to hold and rotate the workpiece,

where its speed depends on the cutting speed. Finally, it is essential for maintaining precision and stability during the machining cycle.

- **Tailstock**: it is positioned opposite to the headstock and its function is to support the other side of long workpieces or hold tools for special mechanical processes (in this case the tailstock is not present on the machine, since the use of drills or reamers for drilling operations have not been taken into account during the project: just considered turning processes and short length workpieces). It can move along the lathe bed to provide additional support.
- **Bed**: it serves as a robust base for all the components of the lathe, the headstock, the tailstock and the carriage. With its rigidity and stability, it prevents any movement and vibration during the work, which could affect the accuracy and the result. On the bed the linear guides are placed aiming for the movement of the carriage system; they are of rectangular and trapezoidal shapes.
- **Spindle**: located inside the headstock, the spindle is rotated by the motor. It is the primary rotating component on which the workpiece is mounted, and its speed and rotation can be controlled to suit various machining tasks.
- **Chuck**: attached to the spindle, the chuck grips and holds the workpiece in place. The involved chuck is a three-jaw type, guaranteeing a self centering, simultaneous and radial closure and the safe holding of different shapes and sizes of the workpiece.
- **Carriage**: it slides along the bed, and it is responsible for carrying the cutting tool. It favours precise lateral (side to side) and longitudinal (back and forth) movements for cutting. The carriage is activated and controlled by a handwheel (manual actuation) or by a motor (automatic actuation), providing smooth and precise positioning of the tool against the work piece.
- **Lead screw**: a long-threaded shaft running parallel to the bed is crucial for the movement of the carriage. It is the mechanical feature that cinematically converts the rotational movement of the handwheel/motor into a linear motion.
- **Tool post**: mounted on the carriage system, the tool post holds the cutting tool. It allows the tool to be positioned and secured at various angles, providing the flexibility to perform a range of cutting operations. It is an adjustable part of the lathe enabling precise control over the cutting tool's position and orientation.
- **Cross slide**: placed on the carriage, it moves perpendicularly to the bed, and it is actuated by a lead screw moving along trapezoidal guides. With the carriage feed's mechanism, it completes the adjustments in the positioning of the cutting tool: it provides movement towards or away from the workpiece, crucial for depth accuracy in cutting operations.
- **Compound rest**: it allows the angled cuts and fine tool adjustments, sitting atop the cross slide. It is substituted with the joint movement of the carriage and the cross slide to machine angular and more complex tasks.
- **Apron**: in the manual lathe it includes the two gears, one for the longitudinal motion and the other for the transversal, respectively for the carriage and the cross slide. This system is automatized with two motors.

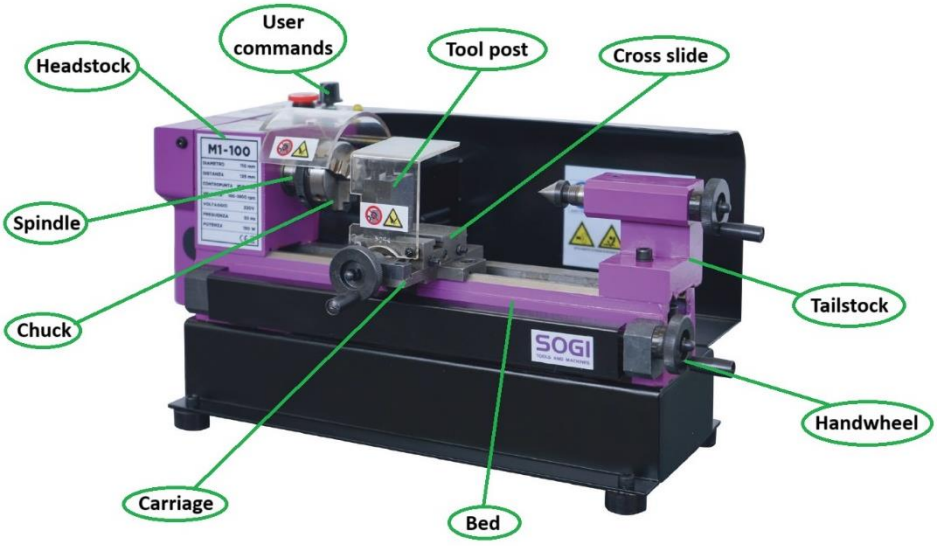


Figure 2: Lathe machine details (1)

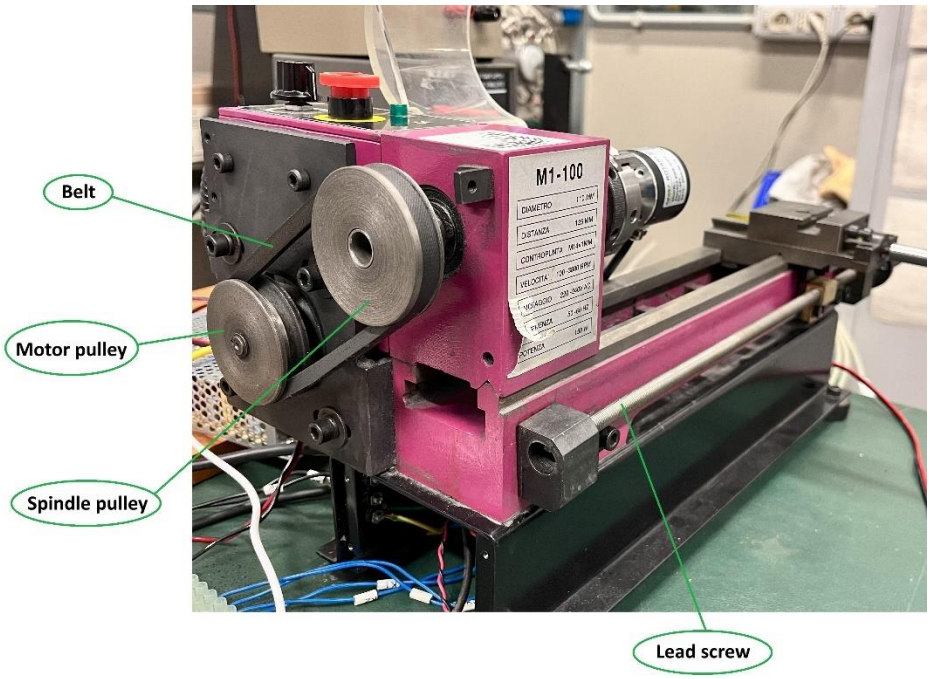


Figure 3: Lathe machine details (2)

A common feature of the lathe machines is their capacity: it is defined as the maximum length and the maximum diameter of the piece that can be turned. The first corresponds to the distance between the tips, and affects the maximum length of the workpiece: for this lathe it is around 200 mm; while the second depends on the height of the lathe axis with respect to the guides.

In industrial lathe with greater dimensions and complexity, the options for the turning processes are very extended, leading to a way flexible and versatile machine. A possible list of them includes the cylindrical and shoulder turning, parting and groove turning, flat facing, drilling and boring, conical turning, external and internal threading, spherical turning, knurling, profiling with form tools and special machining. The variability of the turning processes depends on the combinatorial movement of the carriage, the cross slide and the tailstock.

For this project, only cylindrical and conical turning are considered, where they are aimed at making the surfaces of a piece assume the pre-established diameters.

A cylindrical turning is obtained by transversally approaching the tool to the piece for a distance defined as the depth of cut and advancing longitudinally by another quantity defined as the feed per revolution [02]. Cylindrical turnings, external and internal, always consist of roughing and finishing phases, carried out with different tools, and can be done with manual or automatic feed and with multiple passes, depending on the excess metal to be removed, the hardness of the material and the power of the machine. The roughing passes are performed with robust roughing tools that allow greater cutting depths, reducing the number of passes; instead, the finishing passes are performed with rounded tip, non-vibrating tools, which give a better surface roughness. The same approach and work sequence is assumed for the conical turning, where the tool must follow the cone profile dictated by the conicity.

As other mechanical industrial machines, their automation is widespread nowadays and also for the lathe machines, defined numerically controlled (CN) parallel lathes.

In lathes that use numerical control, all the information relating to the machining operations is associated with a series of instructions that constitute the work program [03]. The control unit, equipped with a microprocessor, interprets the program instructions and transforms them into control signals that it sends to the actuators. NC lathes are equipped with transducers that transform the relative controlled quantities (position, speed, etc.) into electrical signals to be sent to the control unit which, in turn, compares them with the control signals. The structure of the machine has also adapted, equipping itself with special rolling or sliding guides with the interposition of turcite sheets to reduce friction and the use of recirculating ball screws with automatic clearance recovery. Another important feature of these machines is the automatic change (controlled by the program) of the tool which, once the machining operation is completed, is replaced by the next one which, in turn, is taken from a tool magazine installed on the machine; there are warehouses equipped with memory and tool recognition devices that can be housed randomly (random warehouse management). Automatic piece change can also be provided with the use of a rotating slide with multiple stations, which allows the assembly of the next piece while the machine is working on the previous one. The fundamental characteristics of numerically controlled machines are the reduction to a minimum of dead times for transferring tools from one process to another and the perfect repetitiveness of the processes on different pieces, even performed at a distance of time; furthermore, with the complete automation of the process, the elimination of processing waste and the total interchangeability of the pieces, even machined in different places are achieved.

Differently from the manual actuated machines, in the CNC machine tool the operator takes on the role of programmer and supervisor because, with appropriate codes recorded on a computer medium, it is possible to transfer the design of the piece to be produced to the machine, by means of a program that becomes the link between the project itself and the processes that the machine will have to perform. The CNC, or control unit, is the set of the computer and the control program installed in it, whose task is to transform the data in code of the piece to be produced into a series of control signals for the movements of the various parts of the machine tool, in order to carry out the necessary processes. The CNC machine tool offers the following advantages: consistency over time of the dimensional quality and shape of the

pieces being processed; reduction of cycle times; reduction of costs; greater flexibility, thanks to the possibility of using the machine with the same tools, to quickly perform different processes by simply changing the program.

In numerically controlled machine tools, it is possible to program the machining operations by processing and drafting a list of instructions, written in a language that the numerical control can understand, called a program. It contains all the geometric and technological information necessary to perform the machining. Programming starts from the drawing of the part to be produced, on the basis of which the dimensions and commands relating to the machining cycle are defined. Programming is manual when it is carried out directly by the operator in charge of the machine tool, or automatic when it is computer-assisted. The program is translated into the chosen programming language and then entered the numerically controlled machine tool via the console keyboard or with data media or via connection to the local network or the Internet. The program is read, interpreted and executed automatically by the control unit, which then controls the machine tool in the machining operation.

A general diagram of the CNC machine tool shows the block diagram of the organization of the CNC machine tool. The program is entered into the CNC of the machine via consoles or auxiliary memories. The CNC distributes, via the interface, the commands for the actuation of each axis (direction and speed), for the rotation of the spindle and for all the services (auxiliary commands). Each of these commands is controlled by the CNC via a return information cycle. The cycle begins, for example, with the movement command sent to the X-axis actuation. The servomotor, having received the command, activates the screw and moves the carriage or table in the programmed direction. The movement is detected by the position reader which sends the reading to the CNC, which compares it with the programmed one. In addition to the movement, the translation speed is also commanded and controlled (for registration positioning or for work), with a speed detector which measures the rotation speed of the axis to which it is connected. In the event of a difference between the actual speed, detected by the speed detector, and the programmed speed, the CNC issues a command to vary the rotation speed of the servomotor, to adjust the speed of movement of the table to the programmed speed. This automatic control system, interposed between the CNC and the machine drives, is called adaptive control and allows, through detection sensors, to adjust in real time all the control parameters of the movements and speed of the machine to the programmed values. A numerical control can be point-to-point or continuous. Point-to-point control is so defined because the tool operates in well-defined positions (discrete positions) in which it comes into contact with the piece, while, during the movement from one point to the next, it moves away from the piece, so that the trajectory followed by the tool during this movement is not important. In continuous control, on the contrary, the movement of the tool in its displacement between two given points must be controlled exactly both in terms of positions and speed, since the shape of the piece itself depends on it. The machining can take place on a plane, or in three-dimensional form.

The control and servo system of the machine tool is able to make the output quantity assume a predetermined trend to appropriately activate the servomotors, depending on the input quantity (measurements) obtained from the transducers. Control systems can be divided into two categories: open loop systems and closed loop systems. Throughout the project

development, both the structures have been employed based on the different components added to the machine.

In open loop systems the control action is independent of the output signal and no measurement of its quantity is carried out, which is therefore not controlled. In this way the system does not have the possibility of ascertaining whether the controlled element has completed the prescribed motion or not. Consider, for example, a control system of a stepper motor: in this case the quantity to be controlled is the number of steps to be made by the motor. The control system activates the motor with an electrical signal consisting of a predetermined series of pulses sent by the control unit. Once the series of pulses has ended, the system emits a signal to stop the motor that has completed a number of steps equal to that set by the system. A possible variation of the output with respect to the desired value, as in the case in which the motor loses one or more steps, is not acquired at the input by the open loop control, and this constitutes the serious drawback of the system.

In closed loop (or feedback) systems, the control action depends on the output signal; it is continuously measured, and its value is compared with a reference quantity, so as to produce, every time there is a difference between the real output signal and the desired one, a corrective action that brings it back to the desired value. A closed-loop control system can be schematized:

- reaction block: consisting of a transducer that converts the physical quantity to be controlled (speed, position) into a proportional electrical signal and a conditioning block that adapts the signal generated by the transducer to be compared with the reference signal;
- subtractor node: it processes the reference signal and the reaction signal, therefore generating the error signal that carries out the appropriate corrective action;
- actuator control block: it processes the error signal coming from the subtractor node and controls the actuator (motor) to produce the desired output signal.

The operating principle, applied to the case of controlling the rotation speed of a machine tool motor, is the following: once the value of the reference voltage (therefore the rotation speed of the motor) has been set, if the motor tends to slow down, the voltage coming from the reaction block decreases, the error and the voltage at the motor terminals increase, consequently, the motor speed also increases, thus compensating for the initial speed decrease. The regulation of the rotation speed of the tool holder axis in work centres, and of the spindle in lathes, is performed with closed loop control.

Also, for the motion of the tool, commanded by a servo motor, a feedback control diagram can be implemented, guaranteeing that the values of the position and movement speed of each axis coincide with those programmed.

The movement command, given by the control unit, passes to the speed variator which, via the pulse generator, sets in motion the stepper motor (clockwise or anticlockwise rotation) which, connected to the lead screw of the slide, moves it to the right or left. The movement speed, in rapid or working, depends on the frequency of the electrical impulses received by the motor. To check that this speed is exactly the one programmed, a tachometric dynamo is inserted, in axis with the stepper motor, which will produce a greater or lesser electrical voltage depending on the rotation speed of the axis to which it is connected. This voltage is sent back to the speed

variator and compared with the programmed speed. In the event of a discrepancy, the CNC, via the pulse frequency variator, corrects the speed to adapt it to the required value. The movement and direction of the slide are measured by the linear position detector, which transmits the relative position reached to the CNC, which commands the stop of the stepper motor. The speed variator also activates the stepper motor locking brake, in the event of a slide stop or for the reversal of the movement. With this scheme, the position, direction and translation speed of each slide or each moving part of the machine tool are commanded, controlled and measured by the CNC at every instant.

2. Components, descriptions and connections

Before going into the analysis of the procedures actuated to automatize the lathe machine work, a dutiful description of the instruments employed in the project is needed.

The core part of the automated process is the Arduino Mega2560 control board, which contains the logic useful to the command all the components added to the machine, mainly regarding the motors with their drivers and the parts for the human-machine interface.

Later for each set of instruments, a brief description is shown with the schematic of the Arduino connections and the code instructions for their settings on the Arduino IDE software.

2.1 Arduino Mega2560 control board and functionalities

The Arduino Mega2560 is a very widespread development board, used for the implementation of quite large projects, since its dimensions in terms of pinout and computational power, with respect to other products of the Arduino company [04].

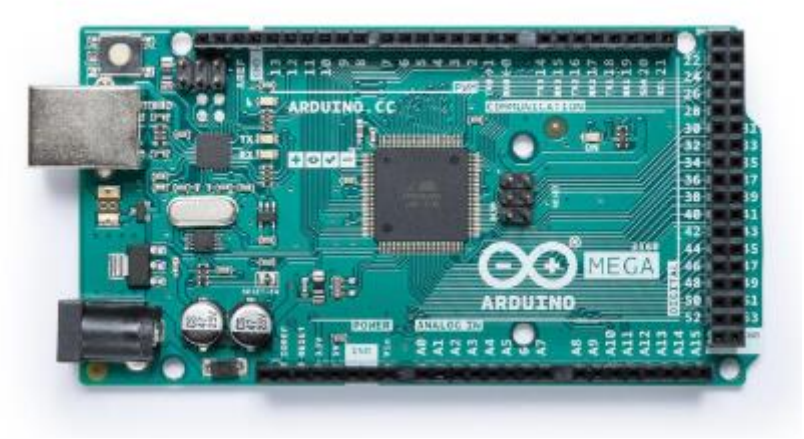


Figure 4: Arduino Mega2560 control board

Most common application fields can be robotic projects or 3D printers, where a lot of connections are required to control various sensors and external devices: for this reason, this thesis work includes two boards of this kind, where motors, drivers, displays, keyboards and sensors must be handled at the same time.

More technically speaking, on the board an ATmega2560 microcontroller is accommodated and a series of 54 digital input/output pins (they supply a constant voltage in time, 0 V or 5 V), 16 analog inputs and 4 programmable UARTs for the serial communication are present. The complete pinout is shown in the picture Fig. 5-6.

The frequency of the ATmega2560 microprocessor is 16 MHz.

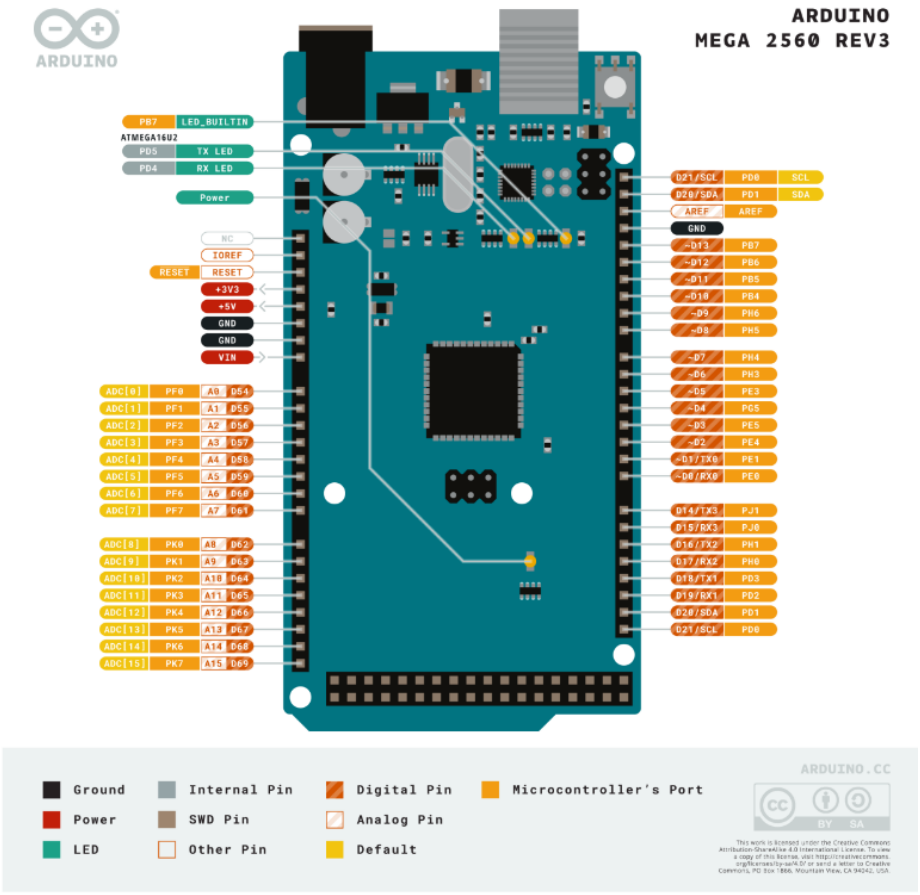


Figure 5: Arduino Mega2560 pinout (1)

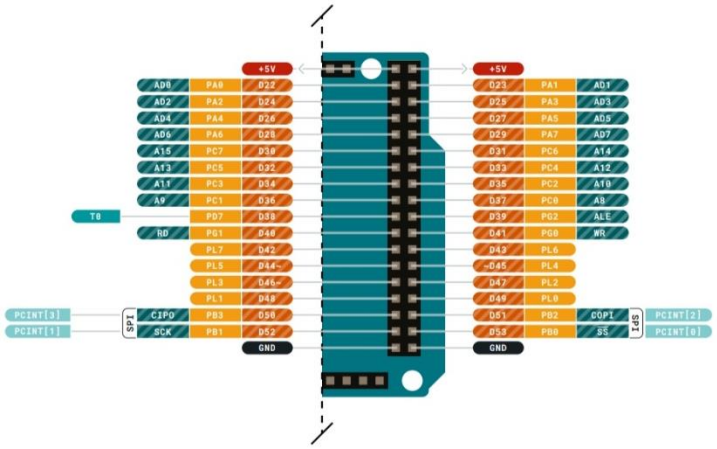


Figure 6: Arduino Mega2560 pinout (2)

Another case of multifunction pins regards the pins from the 54 to 69: they support both input analog and digital input/output signals, and for this work the latter configuration is used.

2.1.1 PWM technique

The Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means [05]. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between the full Vcc of the board and off (Ground) by changing the portion of the time the signal spends on versus the time that the signal spends off. To get varying analog values, the pulse width is changed.

It is a widespread control method, such as for the speed regulation of the motors.

Inside the Arduino IDE, the PWM is recalled with the "analogWrite" function, where its parameters are the output pin of the controller and the value of duty cycle, ranging from 0 to 255.

The output square wave has a frequency depending on the pin chosen, but internally on the timer attached to that pin.

The ATmega2560 processor has 5 timers controlling the 15 PWM outputs with default frequencies. By manipulating the timer registers directly, more flexibility and control can be achieved.

Each timer (Timer from 0 to 4) contains two output compare registers that control the PWM width for the timer's two outputs: when the timer reaches the compare register value, the corresponding output is toggled. The two outputs for each timer will normally have the same frequency but can have different duty cycles (depending on the respective output compare register).

In the simplest PWM mode, the timer repeatedly counts from 0 to 255. The output turns on when the timer is at 0 and turns off when the timer matches the output compare register. The higher the value in the output compare register, the higher the duty cycle. In the Fig. 7 two examples of different values inside the output compare registers (OCR), producing two square waves with the same frequency but various duty cycle.

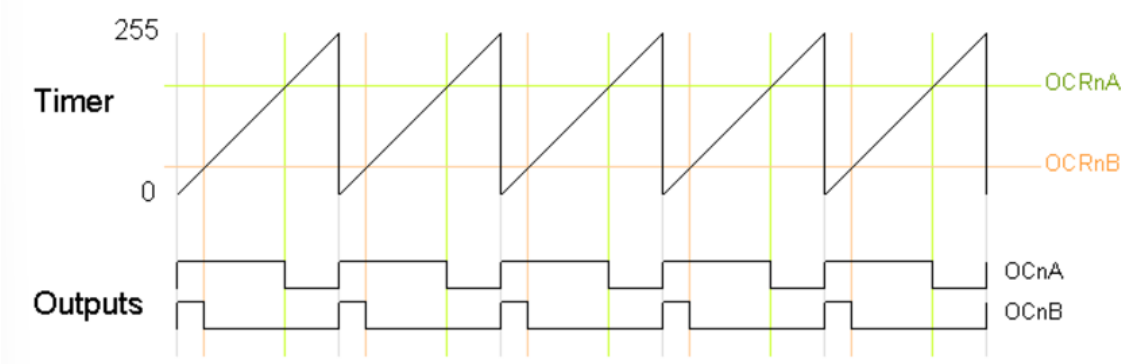


Figure 7: PWM output examples

The default frequencies of the timers are 62500 Hz and 31250 Hz: they are acquired by the base frequency of the microprocessor, 16 MHz, divided respectively by 256 (result 976 Hz) and 512 (488 Hz). Instead, the default frequencies of the PWM outputs are the above values divided by 64, that is the standard value of the prescaler register. There exists a series of possible prescaler values useful to change the PWM frequency: in order to obtain this variation, a bit operation

must be performed through a suitable code (code 1). For the purpose of this thesis work, this modification has been made especially for the PWM pin n° 6, attached to the timer n° 4, restoring the higher frequency of 31250 Hz (prescaler value equal to 1). The code 1 shows the precise commands useful for this purpose.

In Fig. 8 and Fig. 9 two examples of PWM outputs with different values of duty cycles produced by the input voltage coming from a 0-5 V potentiometer: a higher input signal corresponds to a greater number between 0 and 255 and so a greater duty cycle.

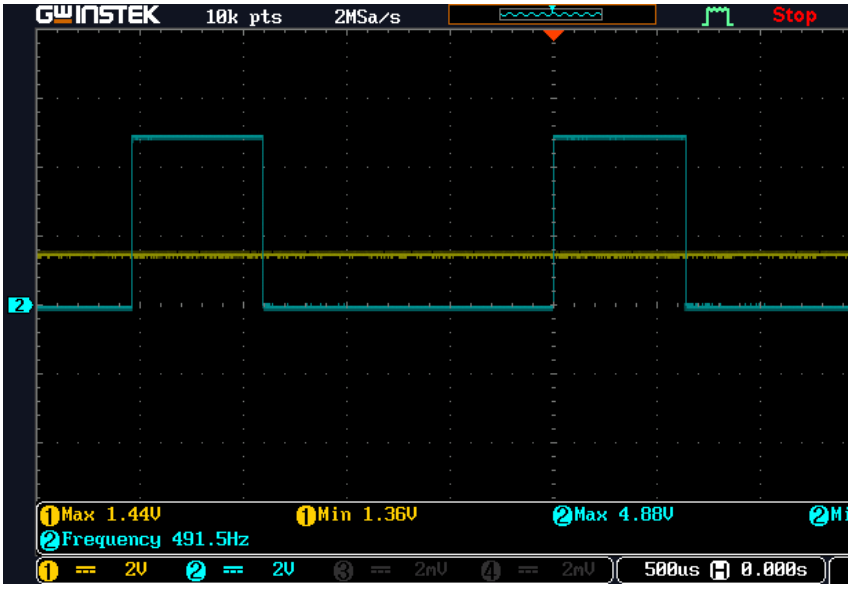


Figure 8: PWM signal at 30% duty cycle and 488 Hz

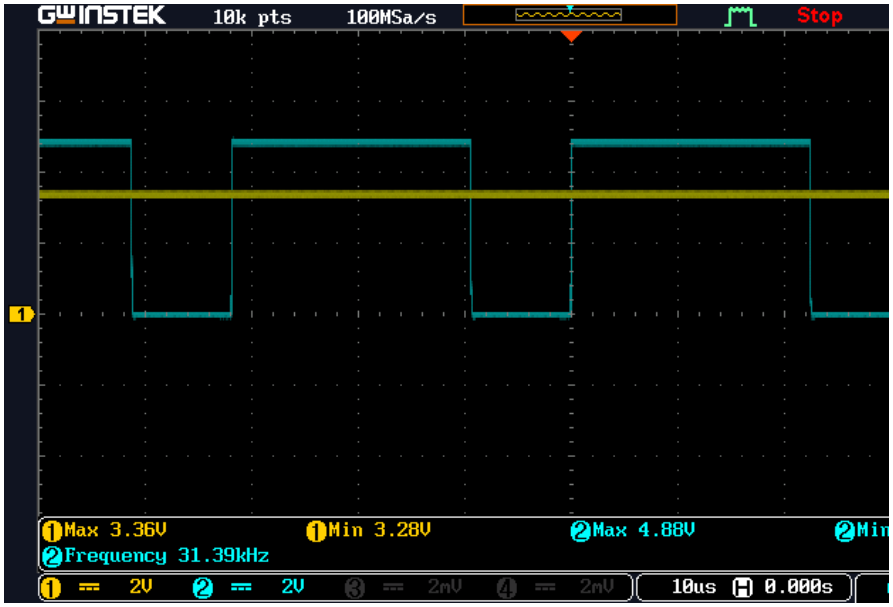


Figure 9: PWM signal at 67% duty cycle and 31250 Hz

Code 1: Timer frequency change

```
#define pwmPin 6 //pin for the PWM signal

void setup(){
  pinMode(pwmPin, OUTPUT);
  setPwmFrequency(pwmPin, 1); //set the pin and the divisor on the timer for the
PWM signal
}

//function to change the frequency of the PWM signal
void setPwmFrequency(int pin, int divisor){
  byte mode;
    //based on the desired divisor, the variable mode assumes an hexadecimal value
    switch(divisor){
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 32: mode = 0x03; break;
      case 64: mode = 0x04; break;
      case 128: mode = 0x05; break;
      case 256: mode = 0x06; break;
      case 1024: mode = 0x07; break;
      default: return;
    }

    switch(pin){
      case 2:  TCCR3B = TCCR3B  & 0b11111000 | mode; break;
      case 3:  TCCR3B = TCCR3B  & 0b11111000 | mode; break;
      case 4:  TCCR0B = TCCR0B  & 0b11111000 | mode; break;
      case 5:  TCCR3B = TCCR3B  & 0b11111000 | mode; break;
      case 6:  TCCR4B = TCCR4B  & 0b11111000 | mode; break;
      case 7:  TCCR4B = TCCR4B  & 0b11111000 | mode; break;
      case 8:  TCCR4B = TCCR4B  & 0b11111000 | mode; break;
      case 9:  TCCR2B = TCCR0B  & 0b11111000 | mode; break;
      case 10: TCCR2B = TCCR2B  & 0b11111000 | mode; break;
      case 11: TCCR1B = TCCR1B  & 0b11111000 | mode; break;
      case 12: TCCR1B = TCCR1B  & 0b11111000 | mode; break;
      case 13: TCCR0B = TCCR0B  & 0b11111000 | mode; break;
      default: return;
    }
}
```

Another key functionality is the serial communication available on the Arduino board. Its main concept is the exchange of data between two hardware devices along cables. There exist several protocols for this communication: the more common are the UART (Universal Asynchronous Receiver-Transmitter) and the I^2C (Inter-Integrated Circuit), both available on the Arduino Mega2560 platform.

2.1.2 UART serial protocol

The first serial protocol analysed is the Universal Asynchronous Receiver-Transmitter [08], which allows the communication and data exchange between the Arduino boards and the computers or other devices: the most common use is the command “Serial.print” allowing the visualization of the data in the Serial Monitor of the Arduino IDE. It performs an asynchronous serial communication in which the data format and the transmission speed are configurable, and so widely used for lower-speed and lower-throughput applications with respect to other serial protocols.

The communication via UART is enabled by the “Serial” class with multiple functions and methods, including the reading and writing data. These last operations can be performed

between devices connected through the RX/TX pins. The table 1 specifies the number of the relative pins for the 4 available channels.

	Serial0	Serial1	Serial2	Serial3
RX	D0	D19	D17	D15
TX	D1	D18	D16	D14

Table 1: UART protocol pinout

And the Fig. 10 depicts the cross connection between two Arduino boards (black for the common grounds connection to define the high and low signals for the communication, yellow TX1 board 1/RX1 board 2 and green RX1 board 1/TX1 board 2).

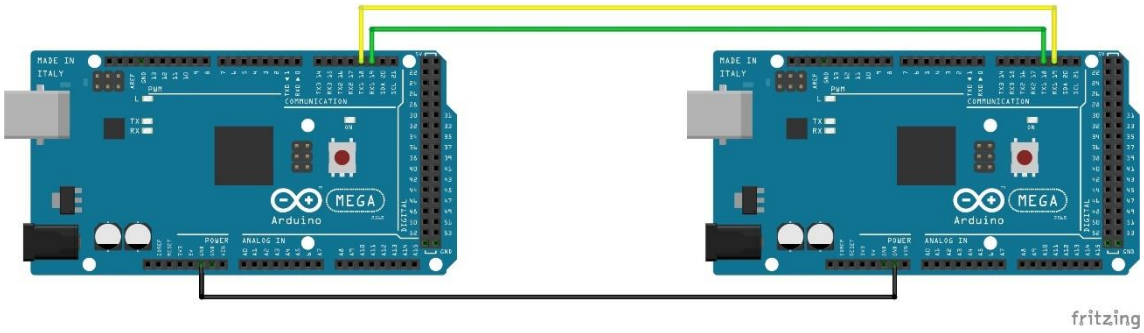


Figure 10: UART serial communication wiring

The real UART transmission operates by sending data as a series of bits, comprehensive of a start bit, the data bits, an optional parity bit and a stop bit. They are sent serially, one bit at a time. As the name suggests, the protocol works asynchronously which means that it does not rely on a shared clock signal, instead it uses predefined baud rates to determine the timing of data bits.

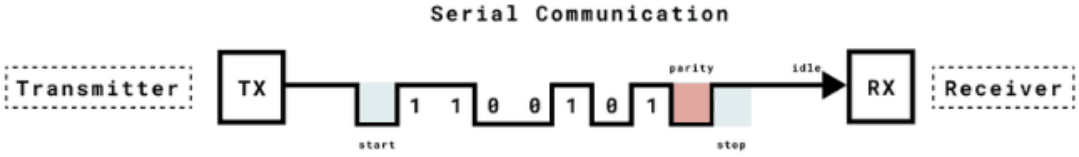


Figure 11: Scheme of the UART transmission

As the Fig. 11 depicts, the transmission requires only one cable for the data passage, so between two devices two wires are needed to accomplish both the data flows (Fig. 10).

Therefore, the figure highlights the two main components, which are the transmitter and the receiver. The former acquires data from a source, formats it into serial bits, and sends it via a TX (Transmit) pin; while the receiver receives the data via a RX (Receive) pin, processes incoming serial data and converts it into parallel data for the host-system.

The last fundamental parameter is the baud rate, that determines the speed of data transmission over the communication channel. It specifies the number of bits transmitted in one second (measurement unit is bits per second, bps), and both the transmitting and receiving devices must agree on the same baud rate to ensure a successful communication.

Its significance lies not only in its direct influence on the speed of data flow, but on its accuracy. A higher baud rate allows for faster data transmission, but it also demands a more precise timing synchronization between the sender and the receiver; on the other hand, a lower value may be suitable for applications where timing accuracy is less crucial with a lower transfer speed. The imposed value is the standard 9600, according to the purposes of the second case.

The real transmission occurs by sending frames of bits, encapsulated by start and stop bits, establishing the boundaries of data transmission and ensuring synchronization between the two devices. The general frame format is the one represented in Fig. 12.

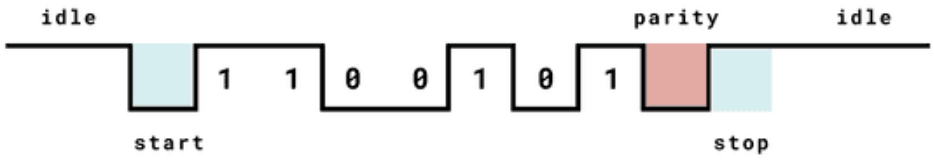


Figure 12: UART frame format (1)

The start bit (Fig. 13) is placed at the beginning of the data frame and its purpose is to indicate the start of the data transmission and prepare the receiver for data reception. It is always logic low (0) for UART (idle state is in high level), meaning that it is transmitted as a voltage level that is lower than the logic high threshold, typically at the receiver's end. When the other device detects a start bit, it knows and prepares to receive the incoming data.

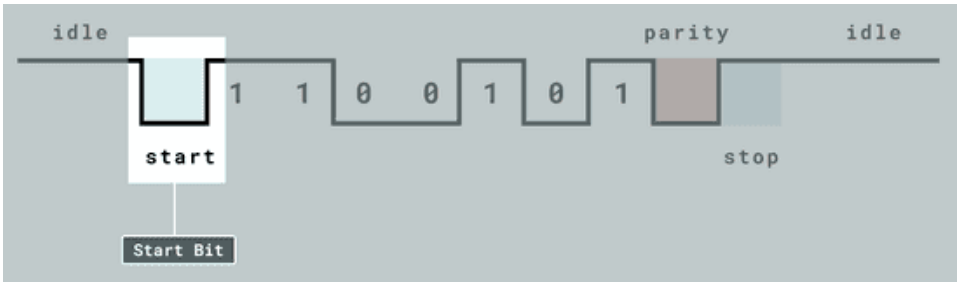


Figure 13: UART frame format (2 - start)

Data bits (Fig. 14) carry the actual binary decoding of the information to be transmitted. Their number inside a UART frame can vary, but a common and widely used configuration is 8 bits, for represents also numerical values. Each bit within the data byte holds a specific position and weight in the binary representation.

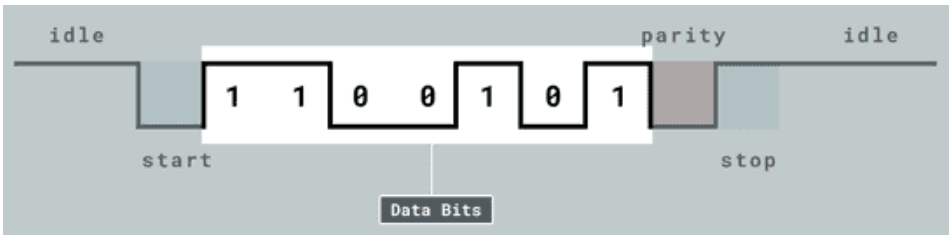


Figure 14: UART frame format (3 - data)

The accuracy of the serial communication relies on the proper configuration of data bits: it is essential that both the transmitter and the receiver agree on the number of data bits and their encoding, because in case of mismatching data corruptions and misinterpretation errors can occur.

After the data bits sequence, a parity bit (Fig. 15) can be part of the UART data frame, and it is an error-checking mechanism helping to detect data transmission errors. Parity can be set to “odd” or “even”, and it ensures that the total number of bits set to logic “1” in a character is either even or odd, depending on the chosen parity type. The presence of a parity bit allows the receiver to verify the integrity of the received data. If the number of “1” bits does not match the expected parity, an error is detected.

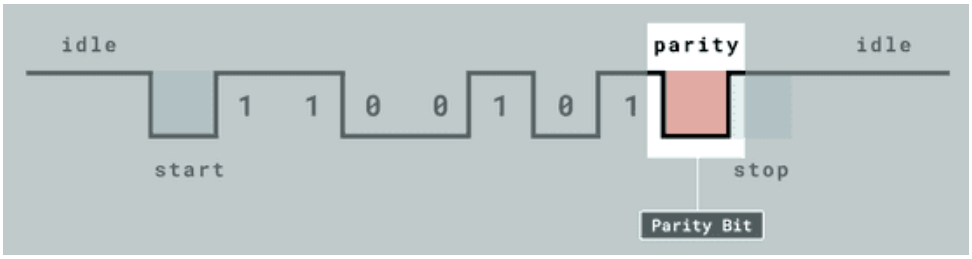


Figure 15: UART frame format (4 - parity)

One or more stop bits (Fig. 16) are sent after the data bits within each UART frame, signalling the end of the data byte and to indicate the conclusion of the data transmission. To a better reliability, two stop bits can be employed. The polarity of the stop bit can vary being high or low level based on the specific UART configuration.

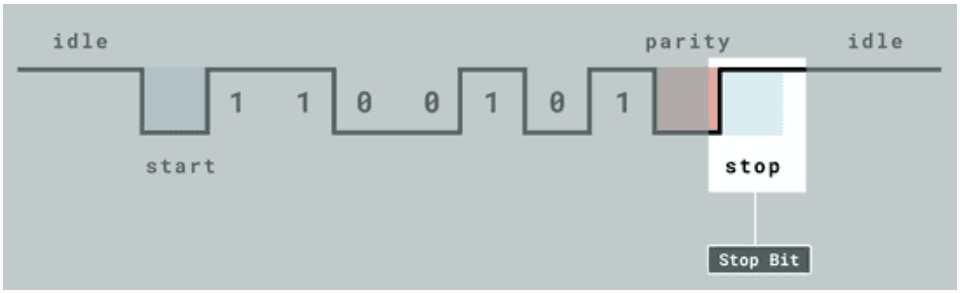


Figure 16: UART frame format (5 - stop)

Finished the serial communication, the logic level of the channel comes back to the idle (high) state, until a new transfer occurs.

A practical application of this protocol takes place during the setup of a turning process, especially for the communication of the theoretical rotational speed of the spindle.

2.1.3 I^2C serial protocol

The second serial communication protocol is the Inter-Integrated Circuit (I^2C) [09], allowing to connect Arduino boards with multiple nature sensors and components with only few wires, favouring flexibility.

It involves using two lines to send and receive data: a serial clock pin (SCL) that the Arduino controller board pulses at a regular interval, and a serial data pin (SDA) over which data is sent between the two devices. In I^2C, there is one controller device, with one or more peripheral devices connected to controllers SCL and SDA lines.

As the clock rises from low to high levels, a single bit of information is transferred from the board to the I^2C device over the SDA line. As the clock line keeps pulsing, more and more bits are sent until a sequence of a 7- or 8-bit address, and a command or data is formed. When this

information is sent - bit after bit -, the called upon device executes the request and transmits its data back, if required, to the board over the same line using the clock signal still generated by the controller on SCL as timing.

Each device in the I^2C bus is functionally independent from the controller but will respond with information when prompted by the controller. This method is based on the master-slave principle: the Arduino board is the master device, while the other devices, such as an LCD display, represent the slavers.

Because the I^2C protocol allows for each enabled device to have its own unique address (identified exploiting a dedicated code), and as both controller and peripheral devices to take turns communicating over a single line, it is possible for the Arduino board to communicate (in turn) with many devices, or other boards, while using just two pins of the microcontroller.

An I^2C message on a lower bit-level looks like in the Fig. 17.

Figure 17: I2C message frame

- The controller sends out instructions through the I^2C bus on the data pin (SDA), and the instructions are prefaced with the address, so that only the correct device listens.
- Then there is a bit signifying whether the controller wants to read or write.
- Every message needs to be acknowledged, to combat unexpected results, once the receiver has acknowledged the previous information it lets the controller know, so it can move on to the next set of bits.
- 8 bits of data.
- Another acknowledgement bit.
- 8 bits of data.
- Another acknowledgement bit.

The SCL clock wire synchronizes the clock of the controller with the devices, ensuring that they all move to the next instruction at the same time.

On the Arduino Mega2560 board, the SDA and the SCL terminals are placed respectively on the pins D20 and D21. The I^2C module requires the supply wires, 5 V and the ground, as well (Fig. 18 for an example).

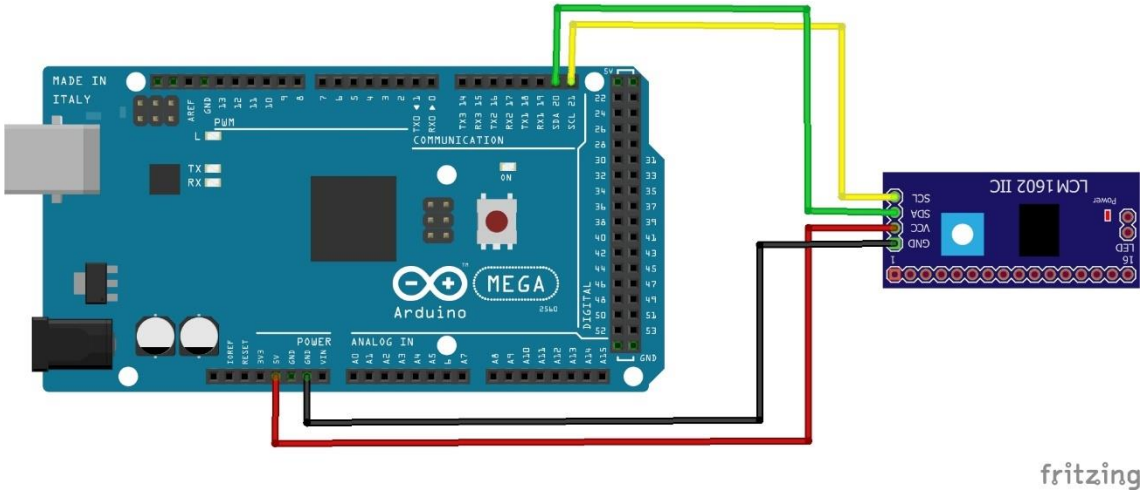


Figure 18: I2C serial communication wiring

2.2 Human-machine interface

In any mechanical machine, the user has to interface with the system even with the more sophisticated and totally automated ones for several reasons, for example safety, to setup the process or to enable and stop the operations on run.

For this project a simple human-machine interface system has been developed, including a keypad and three displays, using different technologies and communication protocols to exploit the differences.

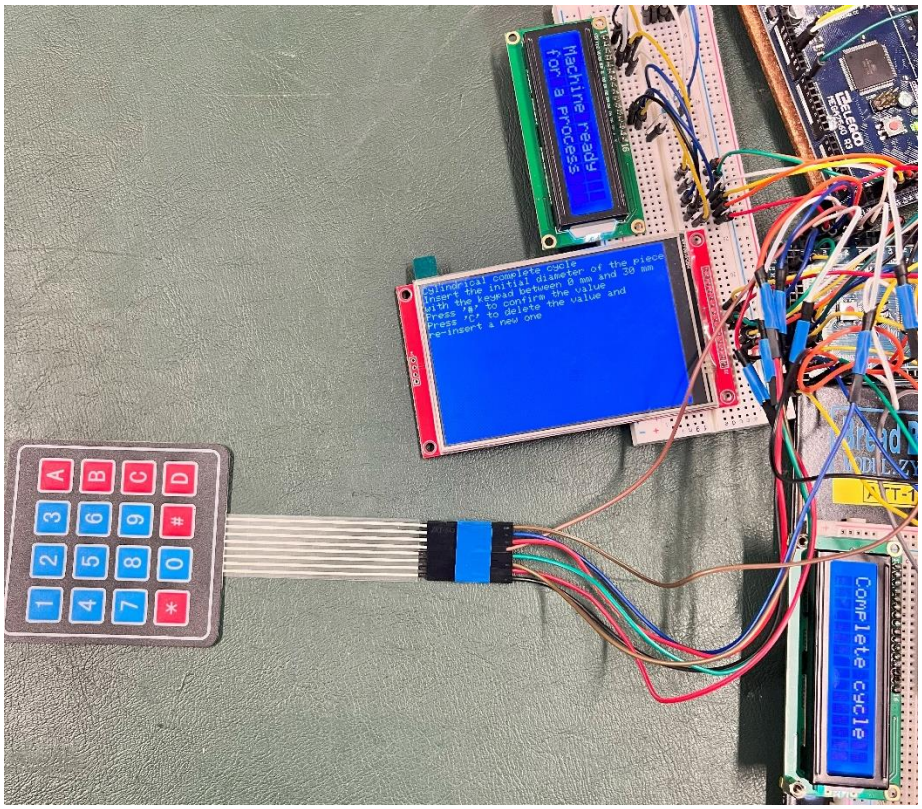


Figure 19: Human-machine interface components and setup

2.2.1 TFT touchscreen display with ST7796S/ILI9341 driver

The main display used to visualize all the information to guide the user during the setup procedure and the showing of the data related to the process evolution, such as the positioning of the tool, is an LCD touchscreen module.

Technically, the display is an LCD module using a 4-wire SPI communication method with a driver IC of ST7796S with a resolution of 320x480 and a touch function [10]. The module includes an LCD display, backlight control circuitry, and touch screen control circuitry.

The table 2 and the Fig. 20 contain the wiring specifications of the display with the Arduino Mega2560 control board. The order of the pins develops from the top to the bottom.

Number	Module Pin	Corresponding to Arduino Mega2560 control board wiring pins
1	VCC	5 V/3.3 V
2	GND	GND
3	CD	A5
4	RESET	A4
5	DC/RS	A3
6	SDI (MOSI)	51
7	SCK	52
8	LED	A0 or 3.3 V
9	SDO (MISO)	50

Table 2: TFT LCD display pinout

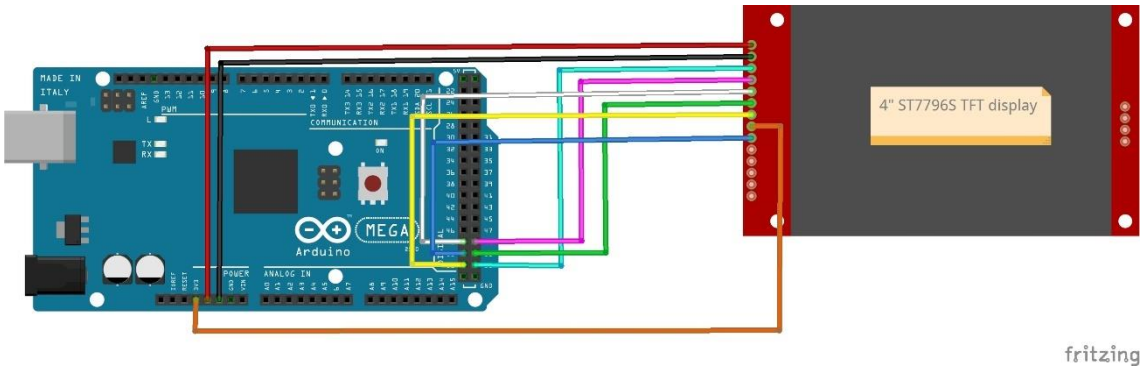


Figure 20: TFT LCD display wiring

The pins allocated for the touchscreen functions have not been exploited, since for the parameters setting by the user a membrane keypad have been connected.

The wiring connections above are referred to the LCD display control circuit, including the control pins and the data transfer pins.

The backlight control circuit handles the state of the backlight to be on or off: if it not required to be adjusted, the backlight control pin can be directly connected to the 3.3 V voltage source without using the control circuit.

Among all the features supported by the ST7796S controller, the technical specifications of this display include the 4-wire SPI (Serial Peripheral Interface) serial ports and 65K RGB colours. The controller uses 16 bits to control a pixel display, so it can display up to 65K colours per pixel. The pixel address setting is performed in the order of rows and columns, and the incrementing and decreasing direction is determined by the scanning mode. The ST7796S display method is performed by setting the address and then setting the colour value.

The data transmission from the Arduino control board to the display exploits the serial communication principles: the data are transferred on a serial channel SDA bit-per-bit (from the most to the least significant bit) according to the time set by the SPI bus clock (each rising edge transmits 1 bit of data). The real data exchange can be performed only when a specific control signal is low, enabling the chip to acquire the data. The last control signal is the data/command control pin of the chip: when this signal is low, the command is written, and when it is high, the data is written.

At code level (code 2), the setting of the display starts including the relative libraries to activate the graphics and the communication between the screen and the microcontroller. Specifically, they are included inside the LCDWIKI library and the ones useful for the project are the LCDWIKI_SPI and LCDWIKI_GUI.

The LCDWIKI_SPI is the underlying library, which is associated with hardware. It is mainly responsible for operating registers, including hardware module initialization, data and command transmission, pixel coordinates and colours settings, and display mode configuration. Instead, the LCDWIKI_GUI is a middle-tier library, which is responsible for drawing graphics and displaying characters using the API provided by the underlying library.

Subsequently, all the required pins are defined in order to generate the display object useful to recall and execute all the functions contained inside the library.

The settings related to the colours accept values in a hexadecimal format: to a human-readable purpose this numbers are saved inside suitable variables.

Code 2: TFT LCD display code setting

```
//settings for the tft display ST7796S
#include <LCDWIKI_GUI.h> // Core graphics library
#include <LCDWIKI_SPI.h> // Hardware-specific library

//parameters definition
#define MODEL ST7796S
#define CS   A5
#define CD   A3
#define RST  A4
#define LED  A0

// This will instantiate (i.e. create) a new LCDWIKI_SPI object
// that you will be able to interact with
LCDWIKI_SPI mylcd(MODEL, CS, CD, RST, LED);

// Assign human-readable names to some common 16-bit colour values:
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
```

```
#define YELLOW  0xFFE0
#define WHITE   0xFFFF

void setup(){
  mylcd.Init_LCD();
  mylcd.Set_Rotation(1); //0 for portrait, 1 for landscape
  mylcd.Fill_Screen(BLUE); //set the colour of the background
  mylcd.Set_Text_Mode(1); //set the orientation of the text on the display
  mylcd.Set_Text_colour(WHITE); //set the colour of the text
  mylcd.Set_Text_Back_colour(BLUE); //set the colour of the text background
  mylcd.Set_Text_Size(2); //size of the text
}

void loop(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Press the command # on the keypad", 0, 0);
  mylcd.Print_String("to start the process", 0, 18);
}
```

Inside the setup environment, all the setting functions are placed, such as the initialization, the background setup and the commands dedicated to the writing procedure (text orientation, colour, back-colour and size).

Finally, an example of writing is added, and the result is pictured in the Fig. 21.

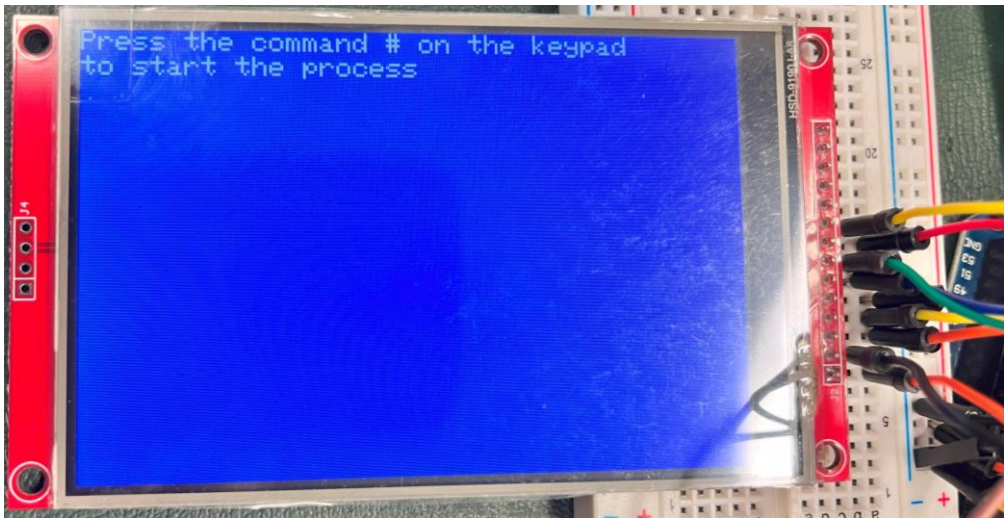


Figure 21: Writing example on the screen

2.2.2 Liquid Crystal Displays and serial communication protocol I^2C

Other components added to the human-machine interface system are two liquid crystal displays (LCD). They have been connected each one to the two Arduino control boards with two different connection methods: one in a direct link and the second exploiting a particular serial communication protocol.

The first one has been linked to the microcontroller handling the speed of the workpiece rotation, visualizing the angular speed measured in rounds per minute. It is connected directly to the Arduino board without any interface with another module, but its pins are linked to the Arduino pins.

The dimensions of the screen are way smaller than the ones of the touch screen, described previously, and so the information sent are limited. Technically, the display is subdivided into 2 rows and each one contains 16 slots, corresponding to maximum 16 characters printed.

The Arduino library available for this product is the “LiquidCrystal” and it takes as parameters the control pins and the 4 bits dedicated to the data [11]. The table 2 lists the pinout of the module and the corresponding pins on the microcontroller (the supply, the control pins and the data pins, which are uncompleted since this display can work also in 8-bit mode). The Fig. 22 shows the wiring: the order of the table coincides with the displacement of the pins from left to right.

Number	Module Pin	Corresponding to Arduino Mega2560 control board wiring pins
1	VSS	GND
2	VDD	5 V
3	V0	GND
4	RS	50
5	R/W	GND
6	E	52
7	DB0	-
8	DB1	-
9	DB2	-
10	DB3	-
11	DB4	53
12	DB5	51
13	DB6	49
14	DB7	47
15	A	5 V through 220 Ω resistor
16	K	GND

Table 3: LCD display pinout

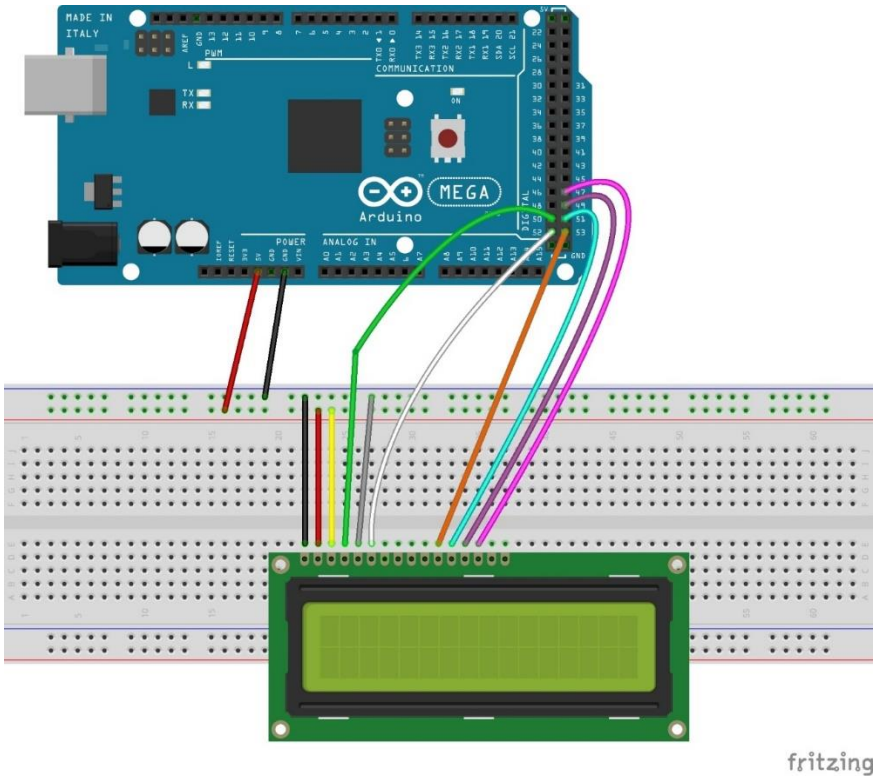


Figure 22: LCD display wiring

The code 3 shows the initialization of the display with the creation of the relative object and list of the main functions used to work with this module.

Code 3: LCD settings

```
//LCD display
#include <LiquidCrystal.h>
const int rs=50, en=52, d4=53, d5=51, d6=49, d7=47;
LiquidCrystal lcdSpeed(rs, en, d4, d5, d6, d7);

void setup(){
  lcdSpeed.begin(16,2);
  lcdSpeed.clear();
}

void loop(){
  lcdSpeed.clear();
  lcdSpeed.home();
  lcdSpeed.print("Speed: ");
  lcdSpeed.setCursor(0,1);
  lcdSpeed.print(String(velFiltInt) + " rpm");
}
```

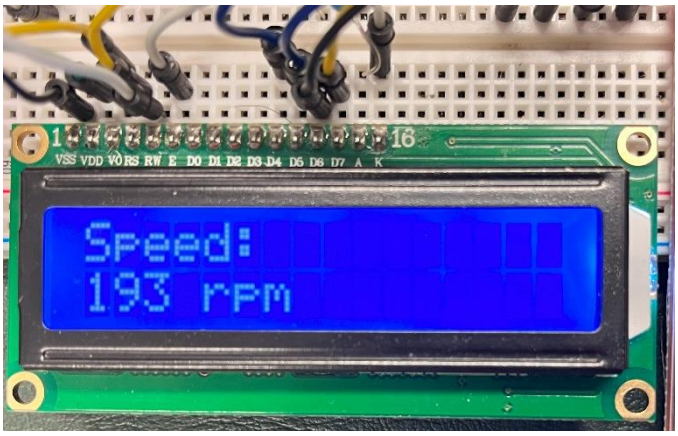


Figure 23: Example of visualization on the LCD

The second LCD has been linked to the main microcontroller, which contains the core program of the automated processes, and visualizes the coordinates of the tool referred to a specific reference frame during its work motion.

The method employed for its communication is the I^2C serial protocol [09]: in order to accomplish this approach a specific interface module must be interposed between the Arduino control board and the LCD.

This "LCM1602 IIC" I^2C module has two sets of connections (Fig. 24): the first includes the 16 pins for the link with the LCD; the second 4 connectors to interface with the Arduino, specifically the power supply (5 V and the ground) and the two peculiar channels of the SDA and the SCL, respectively the data line and the serial clock line. A practical and favourable aspect to use this system is the reduced number of cables attached to the microcontroller.

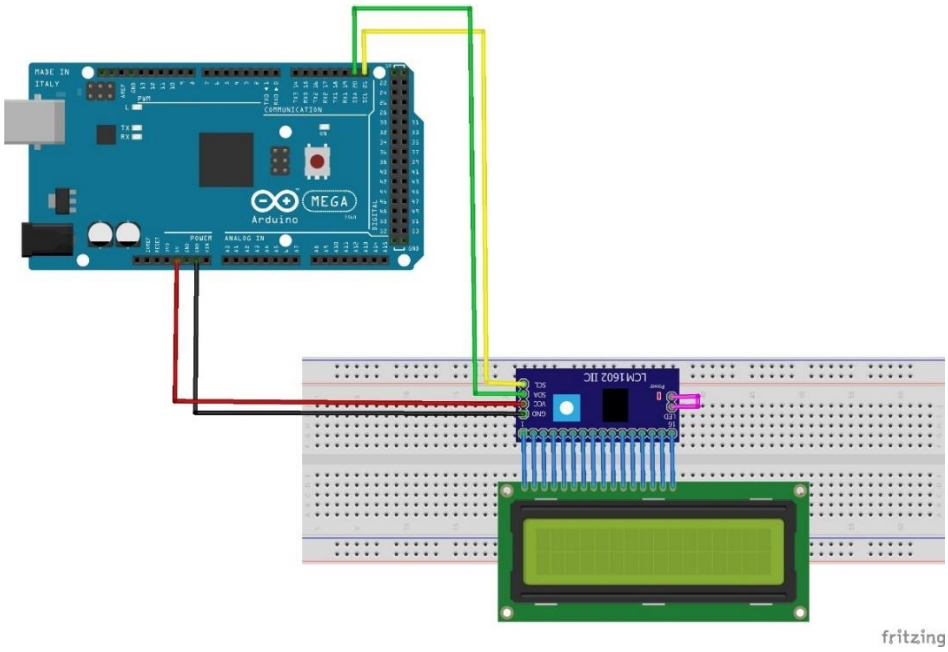


Figure 24: LCD display wiring with I2C module

The setting of the LCD (code 4) is almost the same of the previous case; the main difference lies in the acquisition of the I^2C address of the LCD: this can be performed using a dedicated Arduino

code which scans all the module attached to the microcontroller and its output is the corresponding address.

Code 4: I2C LCD settings

```
//LCD display with I2C serial communication
#include <LiquidCrystal_I2C.h>

#define I2C_ADDR 0x27
#define BACKLIGHT_PIN 3 //default pins for the serial communication
#define En_pin 2
#define Rw_pin 1
#define Rs_pin 0
#define D4_pin 4
#define D5_pin 5
#define D6_pin 6
#define D7_pin 7

LiquidCrystal_I2C lcd(I2C_ADDR,En_pin,Rw_pin,Rs_pin,D4_pin,D5_pin,D6_pin,D7_pin);

void setup(){
  lcd.begin(16,2);
  lcd.clear();
  lcd.setBacklightPin(BACKLIGHT_PIN,POSITIVE);
  lcd.setBacklight(HIGH);
}
```

As said before, on this display the coordinates of the tool are shown during the work cycle. In order to not interrupt the correct work execution, this task has not been developed on the touch screen due to refreshing criticalities, which takes too much time: thanks to its faster communication speed, the LCD represents a good alternative for the purpose. The main TFT screen is used to visualize the static information about the processes.

The Fig. 25 depicts an example of the carriage coordinates visualization.

Figure 25: Visualization example

2.2.3 Membrane keypad 4x4

The last feature useful for the user interface is a membrane keypad [12]: it allows the operator to insert the commands and the parameter values for the execution of the processes.

It consists of a matrix of keys and wires, 4 rows (blue wires in Fig. 26) and 4 columns (orange wires in Fig. 26), with alpha-numerical and symbolic entries.

By pressing one key, the signal relative to the combination row-column becomes up and the information is transferred to the microcontroller.

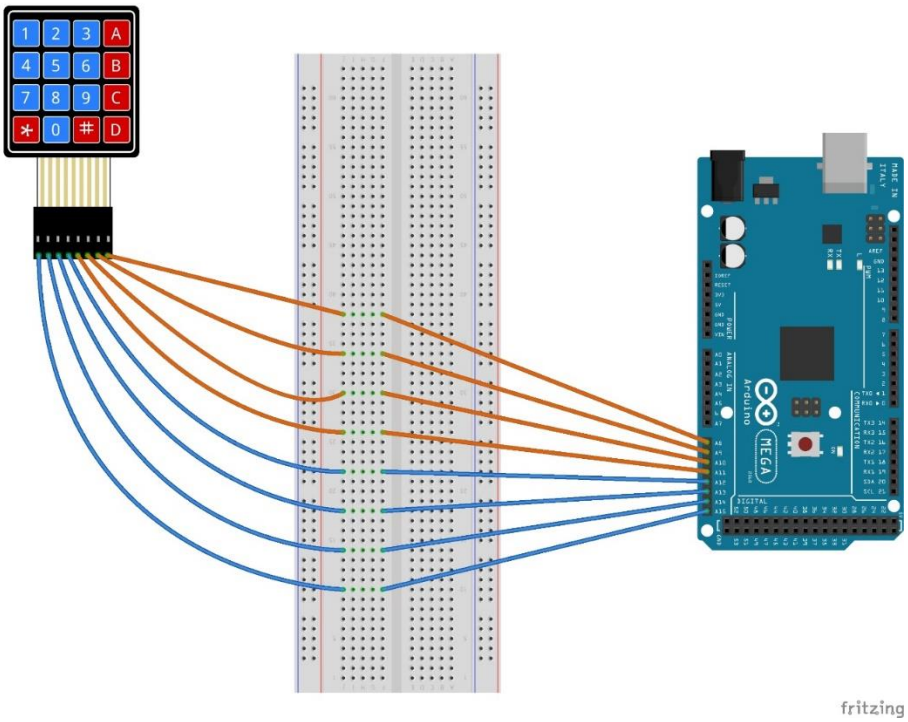


Figure 26: Membrane keypad wiring

The setting at code level of the keypad starts by including the relative library containing the functions able to convert the electric signal coming from the pressing of a digit into a predefined data. The keyboard must be translated into a matrix combining 4 rows and 4 columns: by the combinations of row-column the corresponding value is identified and stored into a suitable variable.

To use the functions, an object is created parametrized by the matrix and by selecting the Arduino pins of the rows and columns.

Once a signal is detected, it is stored inside a char variable and with suitable identification functions the precise meaning is given. A possibility is to use a “switch” structure to associate to each key a specific command: for example, this approach has been developed for the selection of the mechanical operation to machine (see the relative chapter for further analysis).

Code 5: Membrane keypad code settings

```
//Membrane keypad definition
#include <Keypad.h>
const byte rows = 4; //constants for the rows
const byte cols = 4; // and for the columns

//Connections to Arduino board
byte rowPins[rows] = {69, 68, 67, 66};
byte colPins[cols] = {65, 64, 63, 62};

//Matrix to represent the keys on keypad
char hexaKeys[rows][cols] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
//Create the keypad object
Keypad customKeypad = Keypad(makeKeymap(hexaKeys), rowPins, colPins, rows, cols);
```

```
void loop(){
  char var = customKeypad.getKey();
}
```

2.3 Ultrasonic sensor HC-SR04

During an automatic turning process, the positioning of the tool before executing the real process is crucial. The carriage system must move before longitudinally and then transversely to place the tool near the right side of the work piece. The measure of the first distance can be performed by a distance sensor, placed on the extreme right side of the lathe bed.

In the set of the distance sensors, the ultrasonic type can be a choice and the one employed is the HC-SR04 [13].

By the presence of two ultrasonic transducers, its working principle is based on the transmission of ultrasonic sound pulses hitting the object mounted on the chuck and listening to the reflected wave (Fig. 27): the measured distance will be the positioning length of the tool.

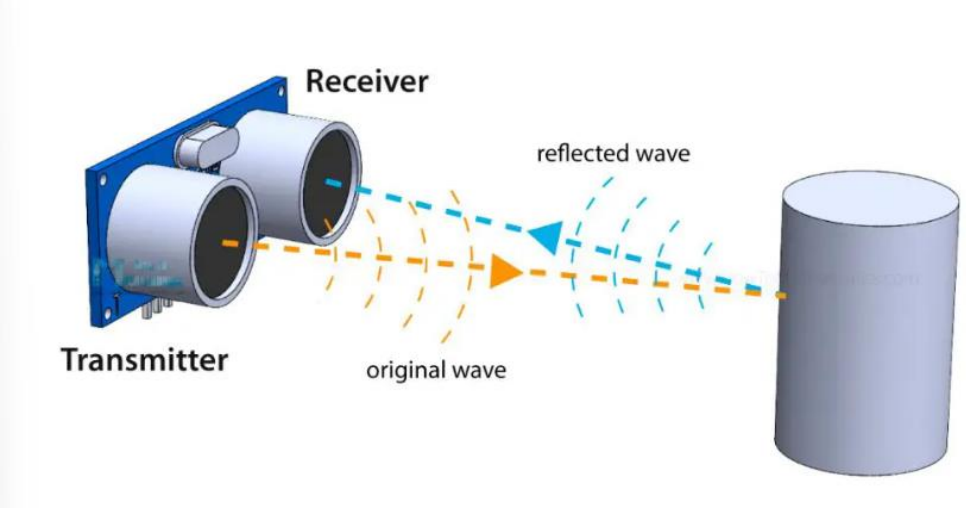


Figure 27: Working scheme of the ultrasonic sensor

The working range is from 2 cm to 400 cm, with an accuracy of 3 mm. About the electrical specifications, the supply voltage is 5 V, the operating current 15 mA and the work frequency 40 kHz.

Fig. 28 depicts the Arduino-sensor wiring: the red and black wires for the voltage supply, while the yellow line for the transmitter link (left – TRIG pin) and the green line for the receiver (right – ECHO pin).

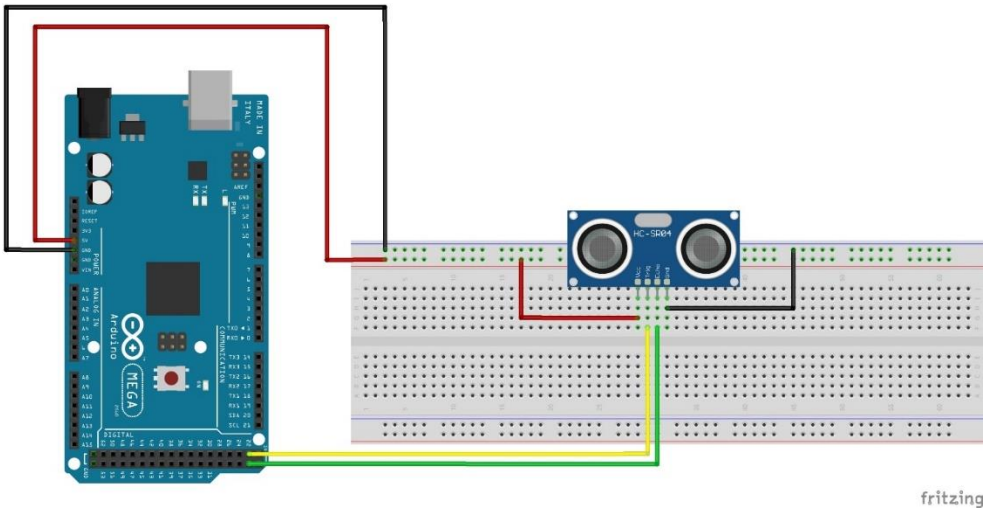


Figure 28: Ultrasonic sensor wiring

The sensor emits an ultrasound at 40 kHz which travels through the air and, if there is an object on its path, it will bounce back to the module. Measuring the travel time and knowing the sound speed, the distance can be calculated.

To generate the soundwave, the TRIG pin is set to the high state for 10 µs: during this amount of time, it sends out an 8-cycle ultrasonic burst which travels at the speed of sound. Subsequently, the ECHO pin goes high right away after that 8-cycle burst, and it starts listening or waiting for the wave to be reflected from an object (Fig. 29).

If there is not any object or reflected pulse, the ECHO pin will time-out after 38 ms and get back to the low state.

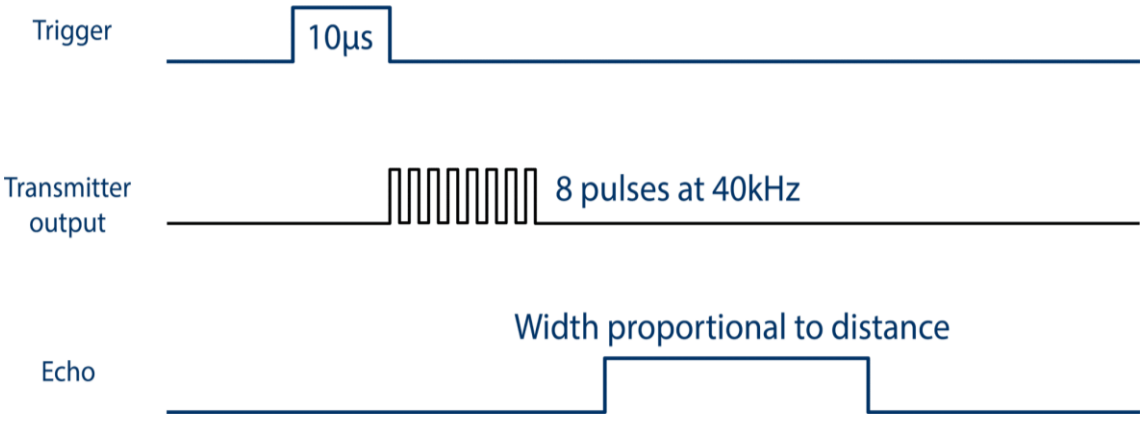


Figure 29: Ultrasound wave schemes

If the ECHO pin receives a reflected pulse, it will go down sooner than 38 ms. According to the amount of time it has been high, the distance sensor-object the sound travelled can be determined by the formula:

$$Distance = \frac{speed * time}{2}$$

Where the variable speed is the speed of the soundwave in the air, which is 343 m/s or 0,343 mm/µs (actual units of measurement used inside the code) and the time is the amount of time the ECHO pin is HIGH. The product result must be halved since it includes both the forward and the backward length covered by the wave signal.

At code level, the definition and usage of the sensor starts with the declaration of the pins involved on the board and the useful variables to store the length value (code 6).

Inside the loop function, an example of distance measurement is shown. With the functions "digitalWrite" and "delayMicroseconds" the soundwave is generated from the transmitter, connected to the "trigPin". The return of the signal is detected by the receiver attached to the "echoPin" translated to an Arduino information with the function "pulseIn": inside the "duration" variable is stored the amount of time the ECHO pin stays in the high state (stop the save when there is the trigger high -> low).

This time quantity is transduced into a distance information exploiting the previous formula with the sound speed.

Before emitting the ultrasound wave, the TRIG pin is set to low to make sure it is in the low state before the operation.

Code 6: Code setting of the ultrasonic sensor

```
//ultrasonic sensor useful to measure the presence and distance of the object to
work with respect to the home position
#define trigPin 22 //generates the soundwave to send to the object
#define echoPin 23 //listen to the soundwave coming back from the object

//variables to store the time interval between the soundwaves and the relative
distance (knowing the speed of the sound in the air)
float duration = 0;
float distance = 0;

void setup(){
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop(){
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  duration = pulseIn(echoPin, HIGH); //acquire the starting instant of time when
the echoPin changes state to HIGH and stop the acquisition of time when the state
changes to LOW
  distance = (duration*0.343)/2;
}
```

The following components are the ones that translate the control signals into mechanical actions and vice versa to generate automatic motions: the stepper motors and the rotary encoder.

2.4 NEMA 17 stepper motor and A4988 driver

The stepper motor is a type of motors widespread in automation world where a position control is required [14]. Another class of motors for industrial applications are the servomotors, which work in a feedback loop configuration with the need of an encoder to regulate the speed to reach and follow a target value. Differently from the previous motor typology, the stepper motor can work in an easier configuration, that is the feedforward control scheme. As the Fig. 30 shows, the target signal for the stepper motor is represented by the position of the output shaft. This signal is fed inside an Arduino controller, elaborated and then sent to the motor driver, in the form of a sequence of pulses to reach that position. The period of these pulses influences the rotation speed of the motor, and their number corresponds to a specific and precise angular position of the shaft.

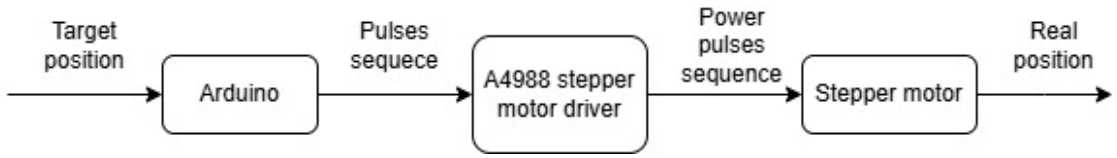


Figure 30: Feedforward control scheme of a stepper motor

Primarily, the sequence is injected into a motor driver, which translates these logic commands into a sequence of pulses of the right power to the coil systems of the motor. The supply voltage of the motor is 12 V and the current flowing across the coil is around 1 A (it can be adjusted with a potentiometer place on the driver board).

Since the actuation method is not based on continuous quantities, but a series of digital commands are sent to the motor, a feedback branch is not required to control the speed of the output shaft: to a certain number of pulses, corresponds a precise positioning of the shaft.

The reasons behind the stepper motor choice concerns the simplicity of the control method without the need of an encoder, leading to a simpler hardware system (just the presence of a power supply and a suitable driver for the stepper motor). The absence of the feedback branch represents at the same time the simplicity reason of the stepper motor and their drawback, since any control on the actual position is present.

Furthermore, the aim of the stepper motor regards the reaching of a precise position during the work cycle, that can be achieved with components of this type, because the goal of the turning is to modify the shape of the product along a precise length.

The rotational speed of the stepper is obviously a physical quantity to consider and control, but it is not the main aspect to take care to perform a turning operation. The speed control happens mainly for the handling of the rotation of the work piece.

Technically a stepper motor is a brushless DC motor, and its working principle is based on magnetic fields. As the common DC motors, they include a rotor and a stator, where the rotor is usually a permanent magnet, and it is surrounded by some coils on the stator (Fig. 31).

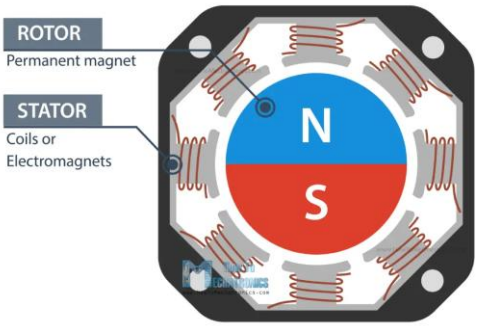


Figure 31: Rotor and stator specification

When a current flows through the coils, particular magnetic fields are generated in the stator that either attract or repel the rotor. By activating the coils, step by step, one after another in a particular order, a continuous motion of the rotor can be achieved, and it can stop at any position. They move in discrete steps.

The resolution and the precision of the motor depends on the number of stopping positions, which can be adjusted by choosing the number of magnetic poles of the rotor. For this project, a NEMA17 stepper motor has been employed. It has 50 stopping points or steps on the rotor and two phases for the coils of the stator, providing four different magnetic field orientations or positions: these lead to have 200 steps per full revolution, or a resolution of 1,8° per step (360° divided by 200).

The four wires to supply the motor, two for each phase, lets the current flow through the phases in both directions.

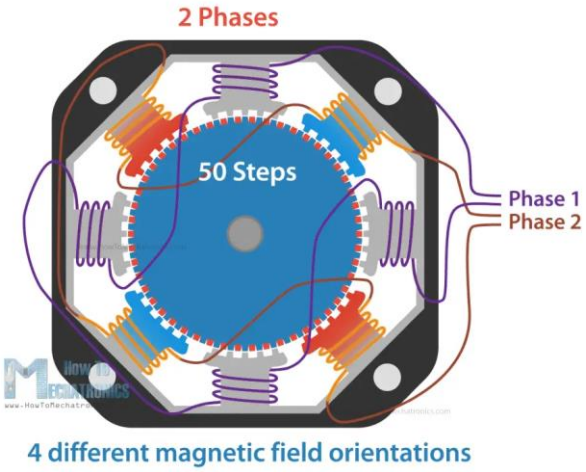


Figure 32: Phases specification

Thanks to this special structure of work, there is the need of energizing the motor phases activating or sending pulses to them in a precise order, in a timely sequence: a suitable driver is required to accomplish that. Their main working principle is that they have two H-bridges (Fig. 33) that allow energizing the motor phases in both directions, providing also other controls such as micro stepping and current limiting. Each H-bridge includes 4 transistor, behaving as switches to allow the passage of the current.

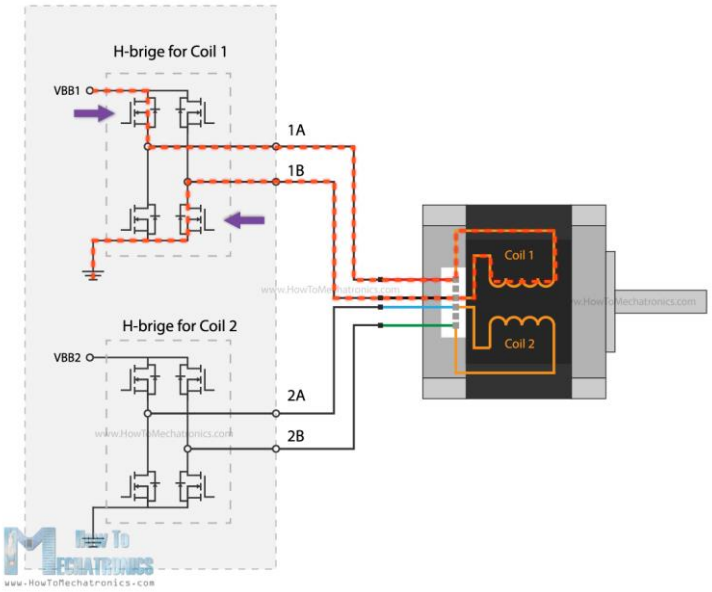


Figure 33: Driver internal circuitry

The chosen driver is the A4988 which satisfies the current requirements to drive the motor, around 1 A per coil.

The Fig. 34 illustrates the overall connections of the driver and the stepper motor with the Arduino control board.

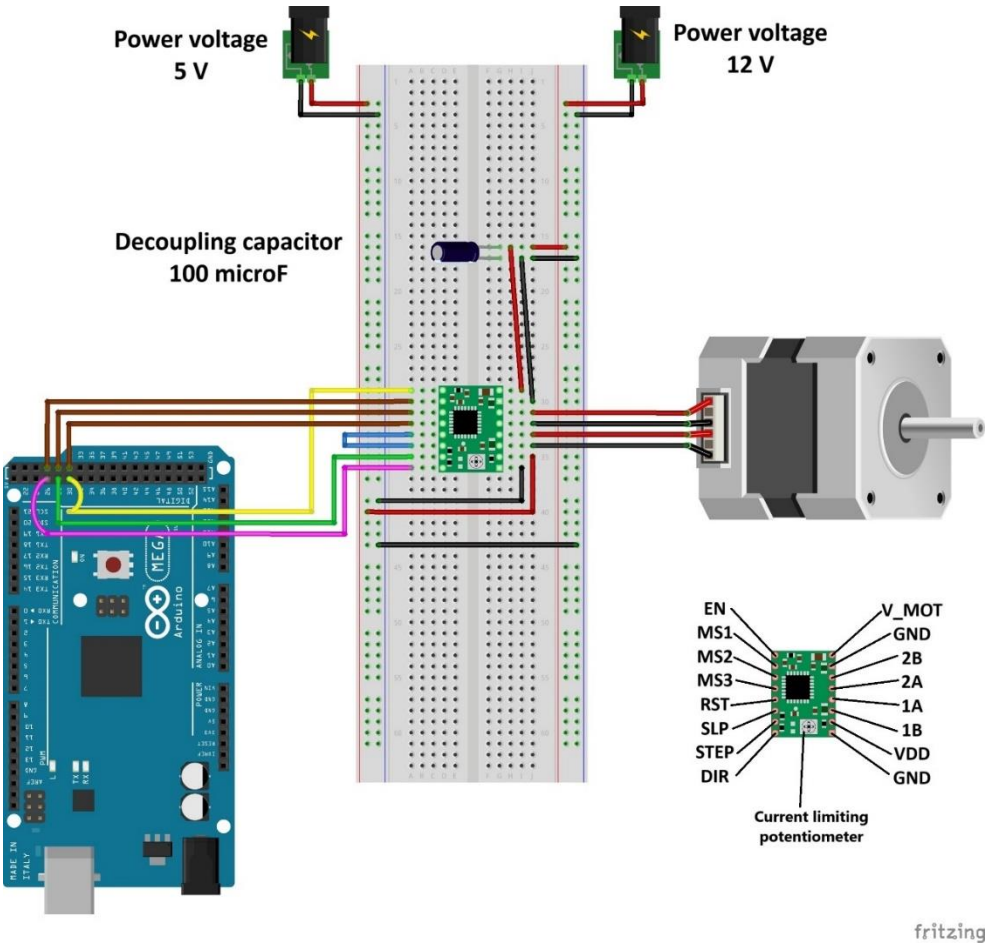


Figure 34: NEMA17 and A4988 driver wiring

The driver board requires two different power sources: the first one is of 5 V, external or directly from the Arduino platform, to supply the logical circuit of the driver, and the second one of 12 V to generate the right power for the phases of the stepper motor.

The four wires of the stator must be jointed to the pins 1A/B and 2 A/B on the right.

On the left side of the board, there are all the control connections. The enable (EN) pin must be set to low, while the reset (RST) and the sleep (SLP) pins can be linked together to enable the driver.

The three pins MS1, MS2 and MS3 are connected to the Arduino to determine the desired number of steps per revolution exploiting the micro step setting. The table 4 outlines the connection combination of the 3 pins, including the number of steps per revolution and the step angle. The configuration chosen for the project is to have 800 steps, resulting in an increased and sufficient resolution.

$$Step\ angle = \alpha = \frac{360°}{number\ of\ steps\ per\ revolution} = \frac{360°}{800} = 0{,}45°$$

Micro stepping

Step mode	Number of steps	Step angle	MS1	MS2	MS3
Full	200	1,8°	LOW	LOW	LOW
Half	400	0,9°	HIGH	LOW	LOW
1/4	800	0,45°	LOW	HIGH	LOW
1/8	1600	0,225°	HIGH	HIGH	LOW
1/16	3200	0,1125°	HIGH	HIGH	HIGH

Table 4: Micro stepping NEMA17 schematic

The last control pins are the step and the direction (DIR): they acquire the digital values to transmit respectively the sequence of pulses to command the motor (logic level to control the transistors inside the H-bridges) and the logic level for the direction of rotation.

All these settings are translated into code language in the extract 7, where the micro step assumed is 800.

Code 7: NEMA17 motor code setup

```
//definition of the connections for the A4988 driver for the stepper motor Z-axis
movement
#define dirPin 26 //pin for the motion direction
#define stepPin 28 //pin for the regulation of the speed of the pulses train
#define enPin 30 //pin for the enable of the driver
#define MS1 27
#define MS2 29
#define MS3 31

//number of pulses to send to the motor to perform one revolution Z-axis
int NpulsesRevolutionZ = 0;

void setup(){
  pinMode(stepPin,OUTPUT);
  pinMode(dirPin,OUTPUT);
  pinMode(enPin,OUTPUT);
  digitalWrite(enPin,LOW);

  pinMode(MS1,OUTPUT);
```

```
  pinMode(MS2,OUTPUT);
  pinMode(MS3,OUTPUT);
  digitalWrite(MS1,LOW);
  digitalWrite(MS2,HIGH);
  digitalWrite(MS3,LOW);

  NpulsesRevolutionZ = 800; //number based on the combination of the 3 pins above
}
```

Inside the lathe machine, this stepper motor is connected to the lead screw performing the longitudinal movement of the carriage and tool.

In order to command the stepper motor with Arduino functions, some preliminary calculations are required.

Firstly, convert the distance to cover into a sequence of pulses to send to the motor, commanding the switching on and off of the coils inside the stator.

$$Angle\ of\ the\ motor = angle\ of\ the\ screw = (number\ of\ pulses) * (step\ angle)$$

$$Distance\ covered\ along\ z\ axis = z = \frac{(pitch\ of\ the\ screw) * (angle\ of\ the\ screw)}{360°}$$

$$Number\ of\ pulses = \frac{360° * (distance\ convered)}{pitch * (step\ angle)}$$

Starting from the desired rotational speed of the shaft, the time elapsed for one rotation is:

$$Period\ of\ one\ rotation = T\ [s] = \frac{2\pi * 60}{shaft\ angular\ speed\ [rpm] * 2\pi}$$

Knowing the number of steps required for one complete turn (800), the half period of the square wave is computed, accordingly:

$$Half\ period\ of\ the\ pulses = T'\ [\mu s] = \frac{60 * 10^6}{2 * 800 * shaft\ angular\ speed}$$

Below an example of the code used to perform one complete revolution of the shaft of the NEMA17 motor. Firstly, the direction of the rotation is defined: HIGH corresponding to the forward direction, while LOW for the backward. Inside the FOR loop, the sequence of pulses is generated according to the frequency computed with the formula above.

Code 8: NEMA17 code period calculation

```
float halfperiodSpeed = 0;
int periodSpeed = 0;

halfperiodSpeed = 60*1000000;
halfperiodSpeed = halfperiodSpeed/(2*NpulsesRevolutionZ);
halfperiodSpeed = halfperiodSpeed/screwSpeed;
periodSpeed = (int) halfperiodSpeed;

digitalWrite(dirPin,HIGH);
for(int i=0; i<pulses; i++){
  digitalWrite(stepPin,HIGH);
  delayMicroseconds(periodSpeed);
```

```
    digitalWrite(stepPin,LOW);
    delayMicroseconds(periodSpeed);
}
```

2.5 28byj-48 stepper motor and ULN2003A driver

The second stepper motor to handle the motion of the tool along the transversal axis is a 28byj-48. As the previous one, its output shaft is linked to the lead screw of the tool post.

The technical performances of this components are lower than the NEMA17, in terms of torque available at the output stage and power supply, in fact it can be directly connected to the Arduino source or an external power module of 5 V.

The purpose of this motor is to position the tool near the work piece according to the feed value and maintain the pose during the process, so a great value of torque is not needed, differently from the longitudinal motor, which must withstand the resistance of the material to be removed. Although the conical process requires the transversal motion of the tool, the movement occurs always perpendicularly to the product.

The physical and the working principle of this motor is quite similar to the previous case, the only difference stands for the number of steps per revolution and the presence of a gear reducer inside the body. In this case, internally are present 4 phases: when one of them is turned on, the internal shaft covers 45°; the reduction ratio is 64, so to complete 360° of the output shaft, 512 sequences of switching on and off of the 4 phases are needed.

The driver useful to convert the logical signal into the right power commands is the ULN2003A: the whole system is depicted in the Fig. 35.

On the left there are 4 wires for the digital command of the 4 phases, below the power supply connections and on the right the links for the stepper body.

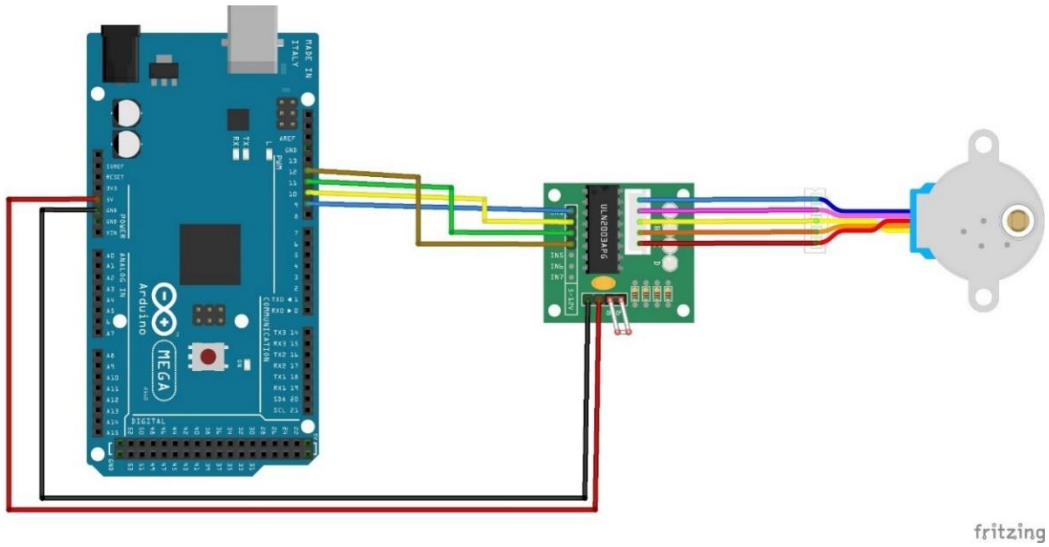


Figure 35: ULN2003A wiring

The code 9 shows the settings for the stepper motor, with the specific functions to supply the coils: to rotate forward, the switching sequence goes from 1 to 4, and vice versa for the reverse direction.

Code 9: ULN2003A code setup

```
//connections of the ULN2003A driver for the stepper motor X-axis movement
#define in1 9 //phase 1 blue
#define in2 10 //phase 2 pink
#define in3 11 //phase 3 yellow
#define in4 12 //phase 4 orange
int delayMotor = 0;

int NpulsesRevolutionX = 512; //full mode for the motor X
const int gearRatio = 64; //gear ratio of the stepper motor in the X-axis
const int angleX = 45; //angle covered by the X-axis motor when a cycle of activation of the 4 phases

void setup(){
 pinMode(in1, OUTPUT); pinMode(in2, OUTPUT);
 pinMode(in3, OUTPUT); pinMode(in4, OUTPUT);

 digitalWrite(in1, LOW); digitalWrite(in2, LOW);
 digitalWrite(in3, LOW); digitalWrite(in4, LOW);
}

void writeMotorX(bool a, bool b, bool c, bool d){
 digitalWrite(in1, a);  digitalWrite(in2, b);  digitalWrite(in3, c);  digitalWrite(in4, d);
}

void Forward(){
 //Step 1
 writeMotorX(HIGH, LOW, LOW, LOW);  delay(delayMotor);
 //Step 2
 writeMotorX(LOW, HIGH, LOW, LOW);  delay(delayMotor);
 //Step 3
 writeMotorX(LOW, LOW, HIGH, LOW);  delay(delayMotor);
 //Step 4
 writeMotorX(LOW, LOW, LOW, HIGH);  delay(delayMotor);
}

void Backward(){
 //Step 4
 writeMotorX(LOW, LOW, LOW, HIGH);  delay(delayMotor);
 //Step 3
 writeMotorX(LOW, LOW, HIGH, LOW);  delay(delayMotor);
 //Step 2
 writeMotorX(LOW, HIGH, LOW, LOW);  delay(delayMotor);
 //Step 1
 writeMotorX(HIGH, LOW, LOW, LOW);  delay(delayMotor);
}
```

2.6 Incremental rotary encoder

The last sensor connected is the incremental rotary encoder, which measures the rotation of a shaft, in order to acquire its angular position and speed. Inside this thesis, it has been linked to the spindle pulley to control the velocity of the workpiece.

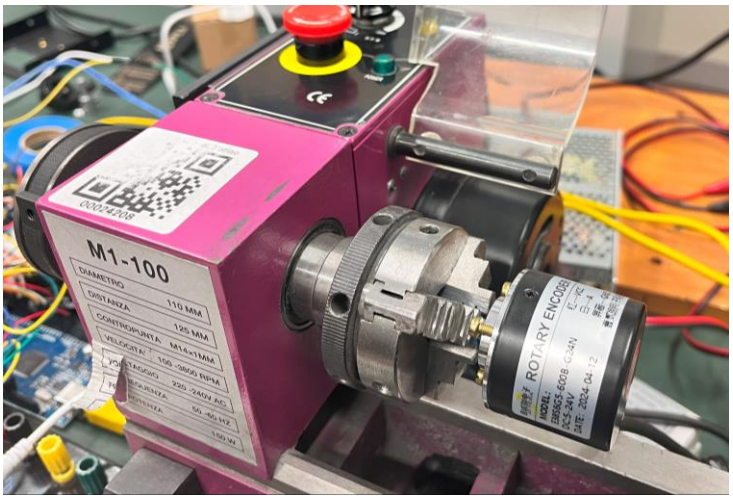


Figure 36: Rotary encoder in a preliminary phase

The output electrical signals of the sensor are two digital square waves, called A and B and between 0 V and 5 V, useful to determine the direction of the rotation.

Inside the encoder there is a disc with evenly spaced contact zones that rotates when the shaft moves. These contact zones are connected to the common pin and to other two separate contact pins A and B. During the rotation an electrical connection occurs step by step between the A and B pins and the common one: the results are two digital voltage signals from the two channels, where the high state is determined by the contact. The two output signals are displaced at 90° out of phase each other.

The resolution of the instrument is 600 pulses generated during a complete shaft revolution.

Initially, by analysing these signals, the position of the shaft is acquired, but measuring the number of pulses per unit of time, the speed is computed.

The connection of the instrument to the microcontroller is quite simple (Fig. 37), based on 4 wires: two are dedicated to the voltage supply (5 V from the Arduino and the ground), while the other two lines for the digital signals generated during the rotation and attached to digital input connectors.

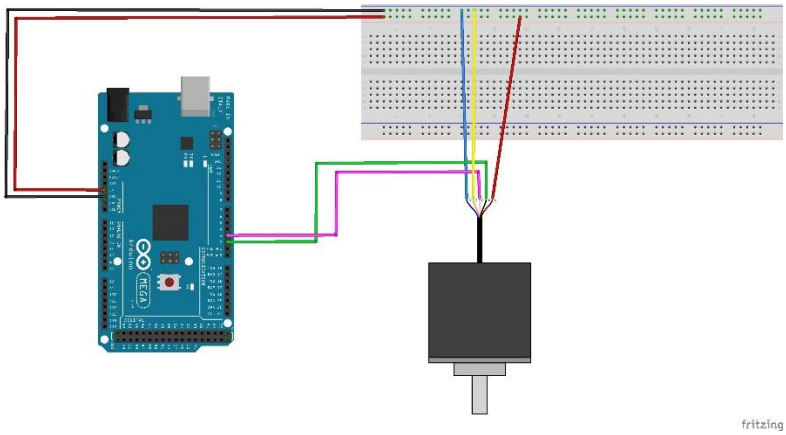


Figure 37: Encoder-Arduino wiring

The Arduino settings of the encoder will be shown and analysed later in the dedicated chapter of work piece speed acquisition and handling.

3. Automation of the working processes

The main working principle of a lathe machine consists of the modification of the dimensions and shapes of various material pieces. Typically, the starting raw workpiece is a cylinder of various diameters. It is mounted, held and fixed on the spindle thanks to three grippers, so during the mechanical work the piece rotate safely not compromising the result.

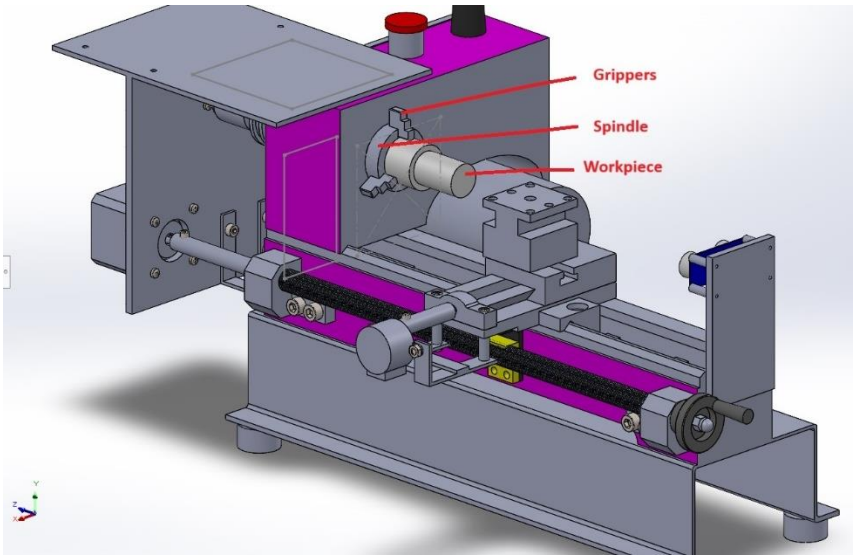


Figure 38: Position of the workpiece on the lathe machine

The component that executes the work by removing the material from the cylinder is the tool: it is a kind of knife which moves alongside the piece. The tool-post holds the tool keeping it fixed with a series of screws, and these two components are set in motion by the leadscrews and the stepper motors. Different cutting shapes in terms of cutting angles and radius exist (Fig. 39), and its choice depends on the outline to be obtained. Also, regarding the tool material, a wide range of different alternatives are present on the market, and again the selection depends on the workpiece material taking into account the economical aspect as well: a balance between the durability of the tool and the quality of the work must be considered.

Figure 39: Examples of different shapes of tools

During the planning of the turning cycle, a series of parameters are involved, regarding all the settings useful to have the best operation and result. These parameters can be divided into two categories: the ones characteristic of the workpiece and the ones proper of the lathe machine.

The former class deals with the sizes of the product. The main concept is that starting from a greater diameter piece, the dimensions are reduced of a certain amount and this change extends

for a pre-established length. In order to translate this information from a mechanical drawing into useful settings for the machine, its work platform is parametrized by a bidimensional reference frame, as shown in the Fig. 40.

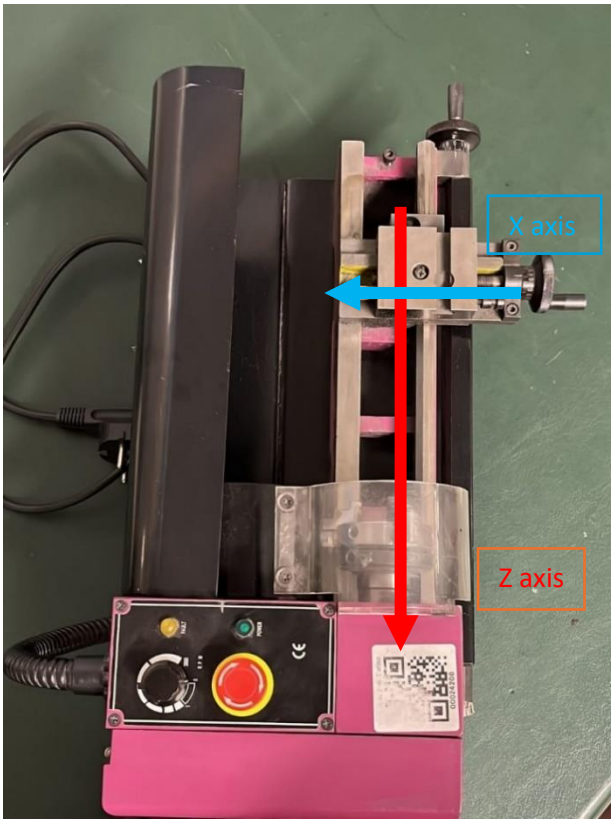


Figure 40: Reference frame of the lathe machine

It includes two axes, displaced along the two possible motion directions of the tool. More precisely, the carriage system is divided into two components, one for each axis and both are moved through leadscrews. For convenience, the two axes are called Z for the longitudinal movement and X for the transversal one: this name rule is acquired by the mechanical manuals. The zero-point of each axis is placed in correspondence of the rest position of the tool during the non-work phases: with respect to the figure, when the carriage is in the top and right position.

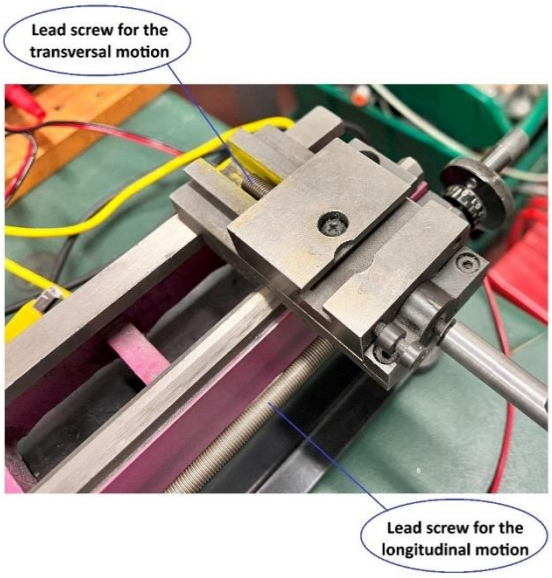


Figure 41: Lead screws detail

In the reference frame, the Y axis does not compare explicitly, because the up-down movement is not allowed, and the tool remains in a fixed position during the work cycle.

As said before, the typical initial shape of the workpiece is a cylinder, but the final shape can be different from a cylinder (Fig. 42). When the input and output appearance of a cylinder is maintained, the turning is called a cylindrical process. Other possibilities can be the screw threading, the conical, spherical or more complex shape turnings: all these operations will modify the profile along both the Z and X directions.

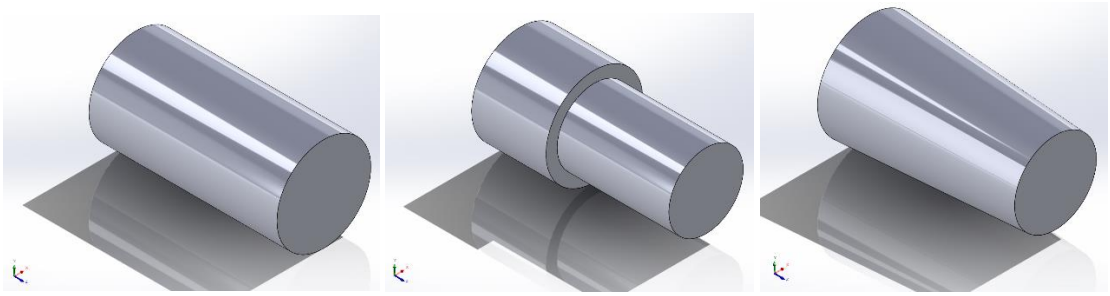


Figure 42: Examples of possible turning products

The parameters required by the process strictly related to the dimensions of the workpiece are generally the start and final diameters and the work length. From these quantities, information about the movement of the tool is acquired: specially by the difference between the two diameters, the work depth is computed, and typically this value refers to the feed along the X-axis, while the length is correlated to the movement of the carriage along the Z-axis.

Starting from these values, the parameters of the second class can be computed. As mentioned before, they refer to the settings for the machine, and more specifically to the speeds of the mechanical parts of the lathe, such as the rotational velocities of the motors and the linear velocities of the tool.

In the next paragraphs, a precise description of each parameter, with the linked mathematical formulation and the code-level translation, will be provided.

A turning process includes several stages, from the setup procedures to finally the execution of the real work, subdivided into a roughing and finishing stage.

Firstly, to prepare the environment for the working cycle, the so-called preliminary operations occur such as the starting on the machine and the choice of the desired work cycle. Once these steps are complete, the piece to work and the suitable cutting tool can be mounted respectively on the chuck and on the tool post placed on the carriage.

Subsequently, all the needed working parameters must be chosen by the operator, following a precise sequence.

Once all the requirements are satisfied to start the work, firstly the positioning phase takes place and then the real practical operation is manufactured.

The main idea to automate the lathe work is to command the movement of the working tool by suitable motors and mechanical joints, linked to the two lead screws. The core element is the Arduino Mega2560 control boards, which acquires, elaborates and generates all the signals coming from or to send to the sensors or motor drivers according to a code written inside the board chip.

To machine the work operation, the communication between two Arduino control boards is required: the first one contains the program including the setup functions, the specifications of the predefined mechanical processes and the command of the tool movement; the other Arduino board handles the rotation of the work piece.

Alongside the normal execution of the process, other contour functions have been implemented, such as the ones for the interface with the operator and an emergency function, in order to handle a sudden misbehaviour of the system.

In the first part, all the common functions to all the 3 processes are developed and described, while later the specific ones are explained.

This preliminary automation has been applied in reality to the machine, as depicted in the Fig. 43 and Fig. 44.

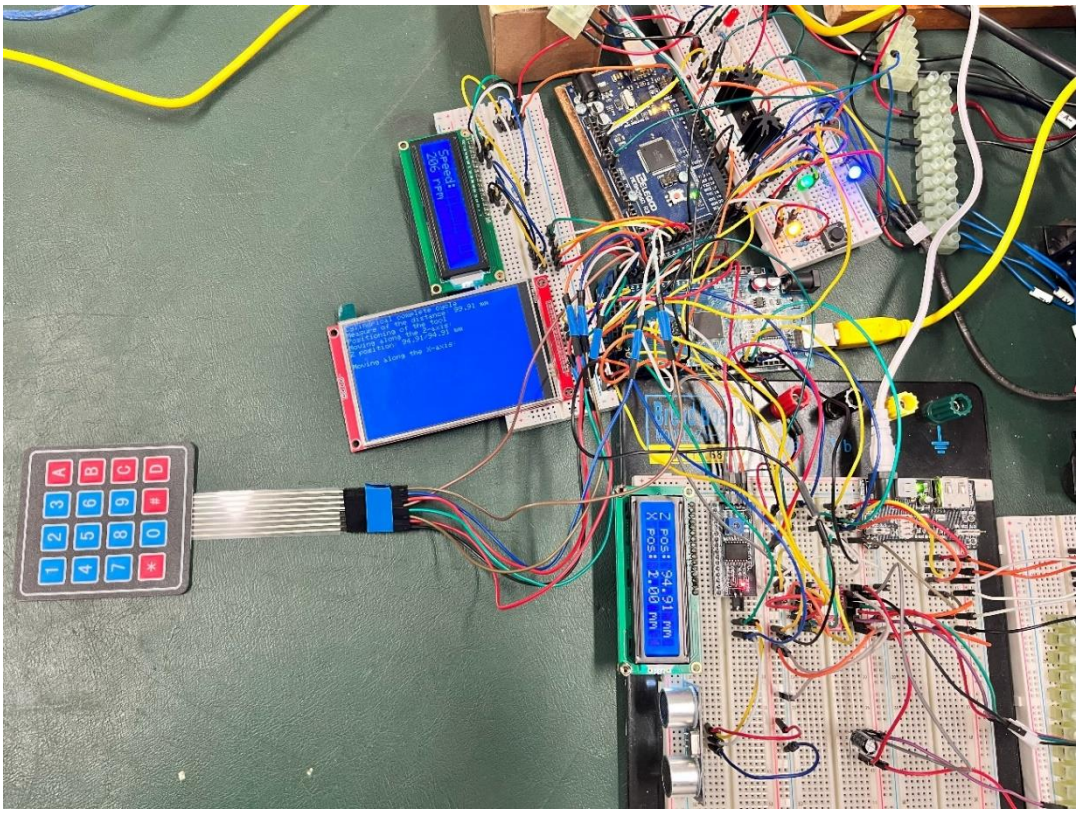


Figure 43: Real application with Arduino boards, sensor and HMI elements

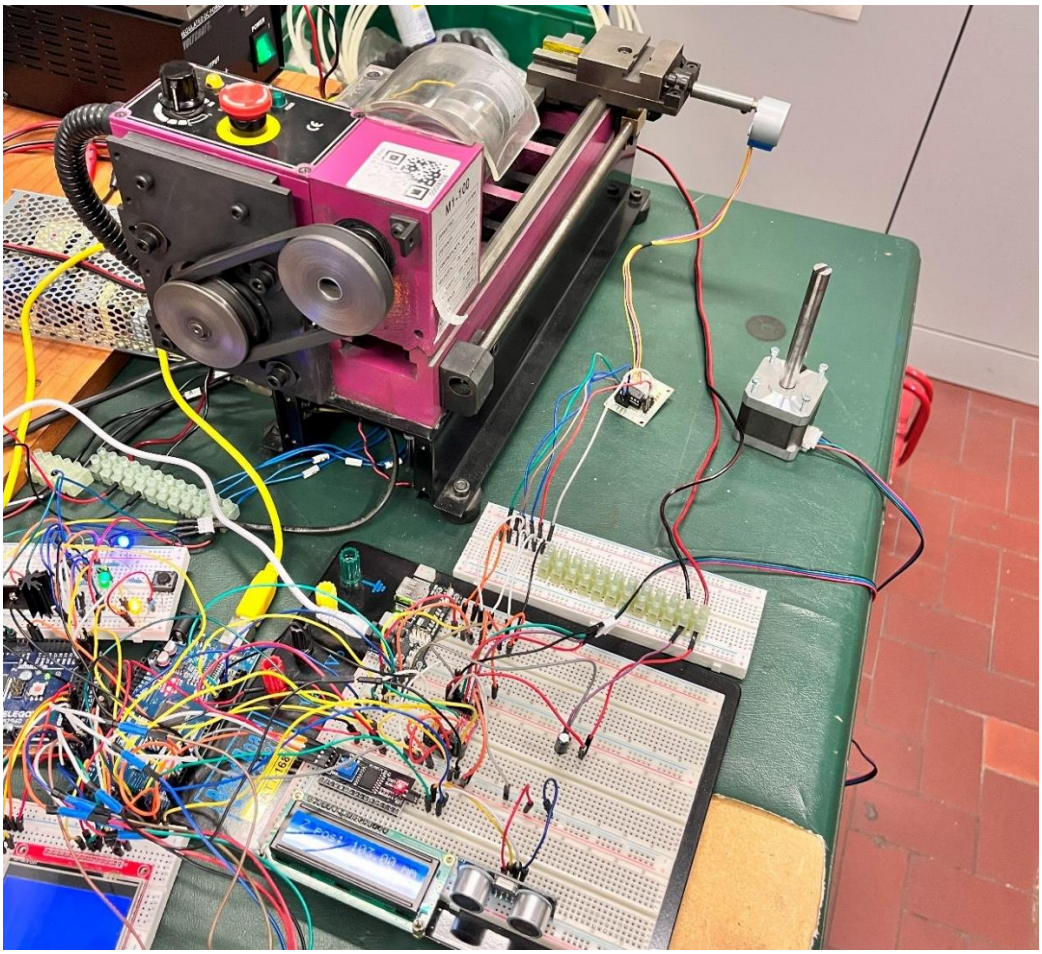


Figure 44: Real application with stepper motors, their drivers and the encoder

3.1 Preliminary operations and human-machine interface (HMI)

The work cycle starts with the preliminary operations, where the user must communicate all the mechanical parameters of the process to the machine through a suitable Human-Machine Interface (HMI).

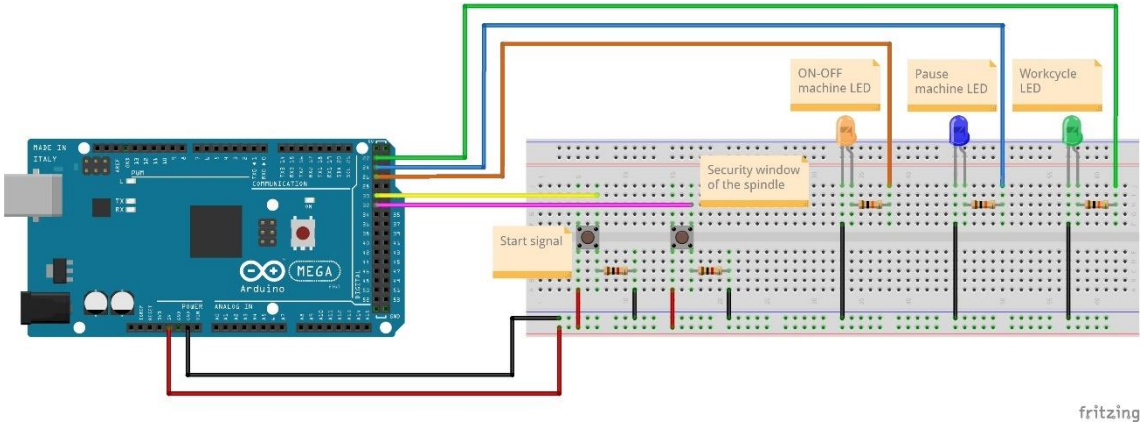


Figure 45: HMI circuits

The Fig. 45 represents the circuits, and the HMI components attached to the Arduino board.

3.1.1 Starting procedure

When the machine turns on (through the check of the ON-OFF signal), the first operation is to enable the start of the process: this occurs pressing a precise key on the keyboard and this can be performed only after the piece has been mounted and secured on the machine.

Figure 46: Enable digit

Code 10: Start signal and preliminary enable

```
#define startPin 34 //pin for the start signal
bool startState = 0; //variable for the state of the start signal

char varStart; //variable to start the process

void setup(){
  pinMode(startPin,INPUT_PULLUP);
  pinMode(secCheckPin,INPUT_PULLUP);
}
```

```
void loop(){
  startState = digitalRead(startPin);
  secCheckState = digitalRead(secCheckPin);

  if(startState == HIGH){ //ON state
    mylcd.Print_String("Press the command # on the keypad", 0, 0);
    mylcd.Print_String("to start the process", 0, 16);

    delay(50);
    varStart = customKeypad.getKey(); //acquisition of the command from the keypad

    if(varStart && varStart == '#'){
      …
    }else if(varStart && varStart != '#'){
      mylcd.Print_String("Insert not valid", 0, 216);
      delay(1000);
    }
  }else if(startState == LOW || secCheckState == HIGH){ //OFF state
    lcd.clear();
    mylcd.Fill_Screen(BLUE);
  }
}
```

The Arduino specific functions that get and check the selection from the keyboard are the "getKey" command and the "if" structure to validate the reading according to the table defined in the variable section.

3.1.2 Process choice

After the enabling, the user must choose the desired working cycle from 3 options, as displayed on the HMI screen:

A. A complete cylindrical turning;
B. A multidimensional cylindrical turning;
C. A conical turning.

Figure 47: Digits and display for the choice of the desired process

The acquisition of the information is done inside a "do-while" function, with the check of the value of a variable as exit condition, and a "switch" function for the selection of the process (code 11). Again, if the insert does not fall inside the set of the ones accepted, a fault is shown, and the program waits for a new input.

Code 11: Process choice

```
char varMode; //variable to choose the operating mode
int ok = 0;

varStart = customKeypad.getKey();
if(varStart && varStart == '#'){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Choose a working process ", 0, 0);
  mylcd.Print_String("from the following list:", 0, 18);
  mylcd.Print_String("> A: complete cylindrical turning with", 0, 36);
  mylcd.Print_String("roughing + finishing stages", 0, 54);
  mylcd.Print_String("> B: cylindrical turning for a", 0, 72);
  mylcd.Print_String("multidiameter component with", 0, 90);
  mylcd.Print_String("roughing + finishing stages", 0, 108);
  mylcd.Print_String("> C: conical turning with", 0, 126);
  mylcd.Print_String("roughing + finishing stages", 0, 144);

  comebackVarZ = 0;
  comebackVarX = 0;

  do{
    ok = 0;
    delay(50);
    startState = digitalRead(startPin);
    varMode = customKeypad.getKey();

    if(varMode && startState == HIGH){
      switch(varMode){
        case 'A': ok = completeProcess();
        break;
        case 'B': ok = multidiameter();
        break;
        case 'C': ok = conical();
        break;
        default:
          mylcd.Print_String("Insert not valid", 0, 216);
          delay(1000);
          mylcd.Fill_Screen(BLUE);
        return;
      }
    }else if(startState == LOW){
      Emergency();
      ok = 2;
    }
  }while(ok == 0);
}else if(varStart && varStart != '#'){
  mylcd.Print_String("Insert not valid", 0, 216);
  delay(1000);
  mylcd.Fill_Screen(BLUE);
}
```

At this stage the rest of the variables for the movement of the stepper motors occur.

As soon as an acceptable working cycle is chosen, the program jumps inside the prescribed operation and all the functions dedicated to the parameter settings can be executed.

3.2 Setup operations

The setup operations regard all the numerical parameters describing the dimensions of the product to obtain and the physical quantities related to the functioning of the machine.

The following descriptions of the stages are proper of the complete cylindrical turning process, since the main logic and procedure is very similar for the other work cycles as well. Further the specific functions proper of each process are explained.

Once the logic enters the prescribed process, the information about the starting of the operation is exchanged with the other Arduino control board through the serial communication (code 12).

Code 12: Enable signal transmission

```
int enb = 0;

void setup(){
  Serial3.begin(9600);
}

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  sendEnableStart();

  ok = 1;
  return ok;
}

void sendEnableStart(){
  enb = 1;
  Serial3.println(enb);
}
```

The reason why it has been chosen to transmit the information, such as the enabling signals and later the other signals, is that with the serial communication exploits only 2 pins and cables, and the same function. With the digital signal exchange, a cable and a pin is required for each one. Instead with the serial transmission, all the data exchange takes place on the same channel and the other unused pins can be exploited for the connection of additional components.

3.2.1 Acquisition of the initial and final diameters

The very first information to be inserted is the diameter of the part under work, before the mechanical operation: this value is saved inside the variable “start diameter” (code 13).

Code 13: Acquisition start diameter

```
float startDiameter = 0.00;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Insert the initial diameter of the piece", 0, 18);
  mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
  mylcd.Print_String("Press '#' to confirm the value", 0, 54);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
  mylcd.Print_String("re-insert a new one", 0, 90);


  startDiameter = acquisitionInitialDiameterB();
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

The value must be obviously set by the user with the keypad.

The acquisition is made inside a “do-while” loop structure, since only one digit of the number can be pressed by the user at a time and read by the microcontroller (code 14). As soon as one keypad signal is detected by the Arduino, this is translated into a certain char variable, if accepted, and stored inside a string. The progressive diameter value is shown on the HMI display.

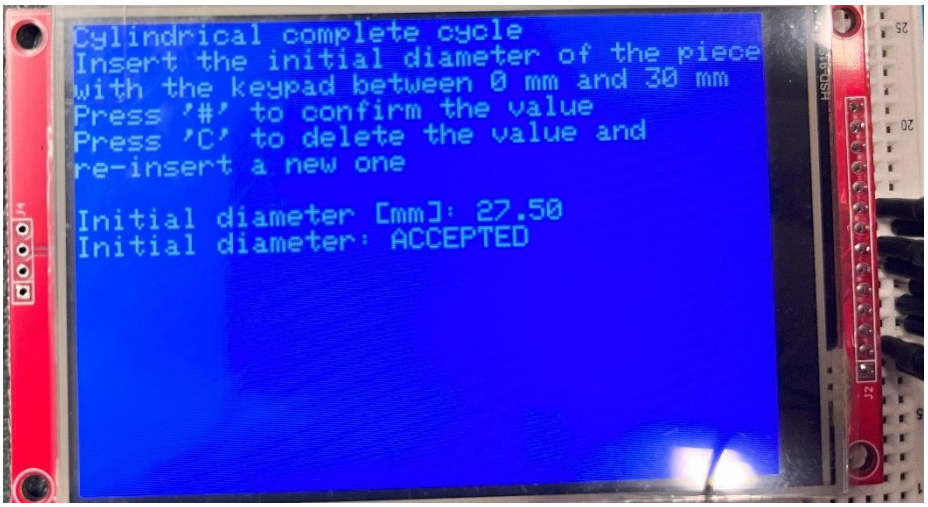


Figure 48: Selection of the initial diameter

A series of checks are required because on the keyboard non-numerical digits can be pressed too: except for the ‘C’, ‘#’ and ‘*’ inserts, the other must be ignored.

Instead, the possibility to delete the value and insert a new one is allowed by pressing the ‘C’ digit, while the confirm of the number happens with the ‘#’, as indicated on the screen. Finally, the ‘*’ is used to insert the decimals of the number.

After the enter signal, the value is converted from a string containing chars into a float variable with the “to Float” function.

As final step, there is the check for the correctness of the inserted value: if it falls inside the admissible range of value, that is from 0 mm to the maximum value allowed (30 mm enforced by the dimensions of the spindle), it is kept, otherwise the acquisition must be repeated until the check exits a positive result.

Code 14: Acquisition start diameter (detail)

```
//maximum acceptable working diameter of the piece [mm]
float maximumDiameter = 30.00;

String diameter1 = "";
bool checkDiameter = 0;

float acquisitionInitialDiameterB(){ //acquisition and check of the initial
diameter
  startDiameter = 0;
  checkDiameter = 0;
  diameter1 = "";

  do{
    delay(50);
    char inCharKey = customKeypad.getKey();
```

```
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      if(inCharKey != 'A' && inCharKey != 'B' && inCharKey != 'C' && inCharKey !=
'D' && inCharKey != '*' && inCharKey != '#'){
        //a number has been inserted
        diameter1 += inCharKey;
        mylcd.Print_String("Initial diameter [mm]: " + diameter1, 0, 126);
        delay(50);
      }else if(inCharKey == '*'){ //'*' = '.'
        diameter1 += '.';
        mylcd.Print_String("Initial diameter [mm]: " + diameter1, 0, 126);
        delay(50);
      }else if(inCharKey == 'C'){
        //clear the variable containing the parameter
        diameter1 = "";
        mylcd.Fill_Screen(BLUE);
        mylcd.Print_String("Cylindrical complete cycle", 0, 0);
        mylcd.Print_String("Insert the initial diameter of the piece", 0, 18);
        mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
        mylcd.Print_String("Press '#' to confirm the value", 0, 54);
        mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
        mylcd.Print_String("re-insert a new one", 0, 90);
        mylcd.Print_String("Initial diameter [mm]: ", 0 , 126);
        delay(50);
      }else if(inCharKey == '#'){
        //all the digits of the diameter and the ok digit have been inserted
        startDiameter = diameter1.toFloat();
        if(startDiameter > 0 && startDiameter <= maximumDiameter){
          //check of the correctness of the diameter
          mylcd.Print_String("Initial diameter: ACCEPTED", 0, 144);
          delay(1000);
          checkDiameter = 1;
        }else{
          mylcd.Print_String("Initial diameter not valid", 0, 144);
          mylcd.Print_String("Insert a new value", 0, 162);
          delay(1000);

          startDiameter = 0;
          diameter1 = "";
          mylcd.Fill_Screen(BLUE);
          mylcd.Print_String("Cylindrical complete cycle", 0, 0);
          mylcd.Print_String("Insert the initial diameter of the piece", 0, 18);
          mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
          mylcd.Print_String("Press '#' to confirm the value", 0, 54);
          mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
          mylcd.Print_String("re-insert a new one", 0, 90);
        }
      }
    }else if(startState == LOW || secCheckState == HIGH){
      Emergency();
      ok = 2;
      return;
    }
  }while(checkDiameter == 0);
  return startDiameter;
}
```

Going through the parameters sequence, now the final diameter must be acquired (code 15): it refers to the final dimension of the piece to be obtained after the chosen turning process.

Code 15: Acquisition final diameter

```
float finalDiameter = 0.00;
```

```
int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Insert the final diameter of the piece", 0, 18);
  mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
  mylcd.Print_String("Press '#' to confirm the value", 0, 54);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
  mylcd.Print_String("re-insert a new one", 0, 90);

  finalDiameter = acquisitionFinalDiameterB(startDiameter);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

The same logic has been applied for this acquisition, updating the suggestions on the display (code 16). A simple check has been added beyond the feasibility of the number, regarding the comparison between this acquired value and the previous one: obviously, since the roughing is a subtraction mechanical procedure, the final size must be smaller than the initial one.

Code 16: Acquisition final diameter (detail)

```
String diameter2 = "";
bool checkDiameter = 0;

float acquisitionFinalDiameterB(float initialDiameter){
  checkDiameter = 0;
  finalDiameter = 0;
  diameter2 = "";

  do{
    delay(50);
    char inCharKey = customKeypad.getKey();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      if(inCharKey != 'A' && inCharKey != 'B' && inCharKey != 'C' && inCharKey !=
'D' && inCharKey != '*' && inCharKey != '#'){
        //a number has been inserted
        diameter2 += inCharKey;
        mylcd.Print_String("Final diameter [mm]: " + diameter2, 0, 126);
        delay(50);
      }else if(inCharKey == '*'){ //'*' = '.'
        diameter2 += '.';
        mylcd.Print_String("Final diameter [mm]: " + diameter2, 0, 126);
        delay(50);
      }else if(inCharKey == 'C'){
        //clear the variable containing the parameter
        diameter2 = "";
        mylcd.Fill_Screen(BLUE);
        mylcd.Print_String("Cylindrical roughing cycle", 0, 0);
        mylcd.Print_String("Insert the final diameter of the piece", 0, 18);
        mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
        mylcd.Print_String("Press '#' to confirm the value", 0, 54);
        mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
        mylcd.Print_String("re-insert a new one", 0, 90);
        mylcd.Print_String("Final diameter [mm]: ", 0, 126);
        delay(50);
      }else if(inCharKey == '#'){
        //all the digits of the diameter and the ok digit have been inserted
        finalDiameter = diameter2.toFloat();
        if(finalDiameter > 0 && finalDiameter <= maximumDiameter && finalDiameter
<= initialDiameter){
```

```
            //check of the correctness of the diameter
            mylcd.Print_String("Final diameter: ACCEPTED", 0, 144);
            delay(2000);
            checkDiameter = 1;
          }else{
            mylcd.Print_String("Final diameter not valid", 0, 144);
            mylcd.Print_String("Insert a new value", 0, 162);
            delay(1000);

            finalDiameter = 0;
            diameter2 = "";
            mylcd.Fill_Screen(BLUE);
            mylcd.Print_String("Cylindrical complete cycle", 0, 0);
            mylcd.Print_String("Insert the final diameter of the piece", 0, 18);
            mylcd.Print_String("with the keypad between 0 mm and 30 mm", 0, 36);
            mylcd.Print_String("Press '#' to confirm the value", 0, 54);
            mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
            mylcd.Print_String("re-insert a new one", 0, 90);
          }
        }
      }else if(startState == LOW || secCheckState == HIGH){
        Emergency();
        ok = 2;
        return;
      }
    }while(checkDiameter == 0);
    return finalDiameter;
}
```

Depending on the final shape, one or more final values are required: exceptions occur for the multidimensional cylindrical and for the conical turnings (specified later in the relative operation chapters).

3.2.2 Definition of the feed along the X-axis

After having defined the start and finish dimensions of the piece, from these the value of feed along the X-axis can be computed: this quantity indicates the depth the tool must cover along the radial direction of the piece, and it is computed generally with the formula:

$$Feed\ along\ the\ X\ axis = \frac{Initial\ diameter - final\ diameter}{2}$$

This information with the working length determines the volume of material to remove.

The feed variable cannot assume any value, but it must fall inside an admissible range, which is defined by the type and material of the working tool: if the depth is too large, the shape of the tool can be warped at the expense of the result of the work. A suitable check has been developed, and so the number of repeats is obtained. The maximum value of feed to determine the number of cycles required is 0,5 mm. If only one cycle is needed (total feed is smaller than 0,5 mm), this means that the final dimension is not so much different from the starting size and a small amount of material must be cut off. While, when the cycles are more than one, during all the repetitions, except the last one, the maximum value of feed is assumed, and in the last turn a smaller distance along the X-axis is covered in order to complete the desired shape.

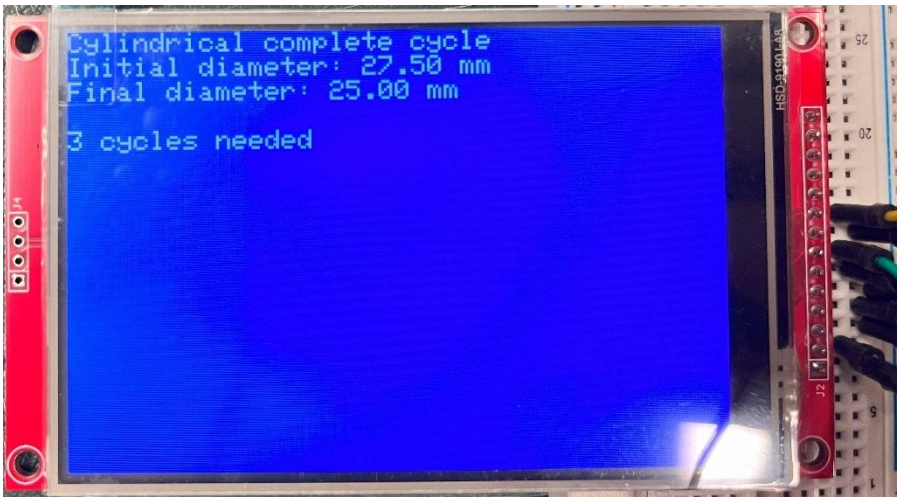


Figure 49: Visualization of the number of cycles needed

Code 17: Standard feed X-axis computation

```
//maximum value allowed for the feed along the X axis
float maxFeedX = 0.5;

float feedXrough = 0.00;
float feedXint = 0;
float feedXfinish = 0.010;
int numberCycles = 0;

int completeProcess(){
  feedXrough = checkFeedXcomplete(startDiameter,finalDiameter);
  delay(1000);
}

float checkFeedXcomplete(float initialDiameter, float workDiameter){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Initial diameter: " + String(initialDiameter) + " mm", 0,18);
  mylcd.Print_String("Final diameter: " + String(workDiameter) + " mm", 0, 36);

  feedXint = (initialDiameter - workDiameter)/2;
  feedXint = feedXint - feedXfinish;
  if(feedXint <= maxFeedX){
    numberCycles = 1;
    mylcd.Print_String(String(numberCycles) + " cycle needed", 0, 72);
    delay(1000);
  }else{
    numberCycles = (int)(feedXint/maxFeedX);
    ver = feedXint - numberCycles*maxFeedX;
    if(ver != 0){
      numberCycles++;
    }
    mylcd.Print_String(String(numberCycles) + " cycles needed", 0, 72);
    delay(1000);
  }
  return feedXint;
}
```

The general feed X value is decreased by a fixed small quantity, addressed to the finishing stage: with the roughing phase the majority of the material is removed, while with the finishing the surface is levelled.

The standard calculation is the one shown above (code 17), applied just for the simple roughing procedure: for the other processes, more articulated formulas will be explained.

3.2.3 Selection of the cutting speed and computation of the spindle speed

Other parameters, fundamental for the good result of the mechanical operation, are the cutting speed and the subsequent angular speed of the part under work.

The cutting speed, expressed in m/min, represents the peripheral speed possessed by the surface of the piece in contact with the tool. This value is chosen mainly according to the materials of the tool and the piece to be machined, but also depends to a lesser extent on other factors such as the section of the chip, the shape of the piece, the lubrication and the surface finish.

Code 18: Cutting speed selection and spindle speed calculation

```
int cuttingSpeed = 0; //[m/min]
int spindleSpeed = 0; //[rpm]

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Insert the cutting speed of the", 0, 18);
  mylcd.Print_String("workcycle with the keypad", 0, 36);
  mylcd.Print_String("between 0 m/min and 40 m/min", 0, 54);
  mylcd.Print_String("Press '#' to confirm the value", 0, 72);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
  mylcd.Print_String("re-insert a new one", 0, 108);

  cuttingSpeed = selectCuttingSpeedB();
  if(ok == 2){
    return;
  }
  delay(1000);

  spindleSpeed = suggestSpindleSpeedB(finalDiameter, cuttingSpeed);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

Specific tables report the value of the cutting speed for the different processes and tool materials, for example for high-speed steel tools the order of magnitude is around 50 m/min, while for the sintered carbide ones 100 m/min. For this project, lower speeds are considered: the acceptable value is between 0 and 40 m/min.

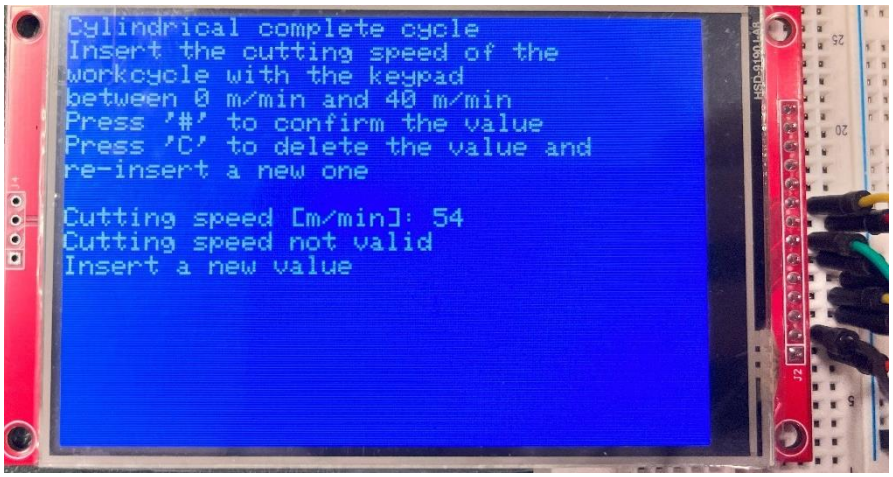


Figure 50: Selection of the cutting speed (error case)

The acquisition of the cutting speed is performed through the keyboard by the user and for the sake of simplicity only integer variables are considered (code 19).

Code 19: Cutting speed selection (detail)

```
bool checkSpeed = 0;
String speed = "";

int selectCuttingSpeedB(){
  cuttingSpeed = 0;
  checkSpeed = 0;
  speed = "";

  do{
    delay(50);
    char inCharKey = customKeypad.getKey();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      if(inCharKey != 'A' && inCharKey != 'B' && inCharKey != 'C' && inCharKey !=
'D' && inCharKey != '*' && inCharKey != '#'){
        //a number has been inserted
        speed += inCharKey;
        mylcd.Print_String("Cutting speed [m/min]: " + speed, 0, 144);
        delay(50);
      }else if(inCharKey == 'C'){
        //clear the variable containing the parameter
        speed = "";
        mylcd.Fill_Screen(BLUE);
        mylcd.Print_String("Cylindrical complete cycle", 0, 0);
        mylcd.Print_String("Insert the cutting speed of the", 0, 18);
        mylcd.Print_String("workcycle with the keypad", 0, 36);
        mylcd.Print_String("between 0 m/min and 40 m/min", 0, 54);
        mylcd.Print_String("Press '#' to confirm the value", 0, 72);
        mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
        mylcd.Print_String("re-insert a new one", 0, 108);
        mylcd.Print_String("Cutting speed [m/min]: ", 0, 144);
        delay(50);
      }else if(inCharKey == '#'){
        //all the digits of the diameter and the ok digit have been inserted
        cuttingSpeed = speed.toInt();
        if(cuttingSpeed > 0 && cuttingSpeed <= maximumCuttingSpeed){
          mylcd.Print_String("Cutting speed: ACCEPTED", 0, 162);
          delay(1000);
          checkSpeed = 1;
```

```
          }else{
            mylcd.Print_String("Cutting speed not valid", 0, 162);
            mylcd.Print_String("Insert a new value", 0, 180);
            delay(1000);

            cuttingSpeed = 0;
            speed = "";
            mylcd.Fill_Screen(BLUE);
            mylcd.Print_String("Cylindrical complete cycle", 0, 0);
            mylcd.Print_String("Insert the cutting speed of the", 0, 18);
            mylcd.Print_String("workcycle with the keypad", 0, 36);
            mylcd.Print_String("between 0 m/min and 40 m/min", 0, 54);
            mylcd.Print_String("Press '#' to confirm the value", 0, 72);
            mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
            mylcd.Print_String("re-insert a new one", 0, 108);
          }
        }
      }else if(startState == LOW || secCheckState == HIGH){
        Emergency();
        ok = 2;
        return;
      }
    }while(checkSpeed == 0);
    return cuttingSpeed;
  }
```

The next step regards the computation of the angular speed the spindle must assume to obtain the best result according to the parameters desired. The theoretical value of this inform is computed according to the following formula:

$$Angular\ speed\ [rpm] = \ n = \frac{1000 * V_{cutting}}{\pi * D_{fin}}$$

Where "$V_{cutting}$" is the cutting velocity, "D_{fin}" is the dimension of the piece after the work (this general formula must be adapted for the more complex cases, such as the multidimensional and the conical turnings, where "D_{fin}" is modified into the average value of the two characteristic dimensions, that are the smaller and the greater diameters).

For a simpler approach dictated by the instruments used, only the integer part has been considered.

Again, a limit check is performed, considering the maximum value acceptable, set to 280 rpm: this is imposed by the power supplying the electric motor actuating the chuck.

Figure 51: Selection of the cutting speed and relative spindle angular velocity

Code 20: Spindle speed calculation (detail)

```
//maximum spindle rotational speed [rpm]
int maxSpindleSpeed = 280; //[rpm]

bool checkSpindleSpeed = 0;
const float piGreco = 3.14;
float toleranceSpeed = 0.02;
int targetMin = 0; //[rpm]
int targetMax = 0; //[rpm]

int suggestSpindleSpeedB(float workDiameter, float cutSpeed){
  do{
    checkSpindleSpeed = 0;
    spindleSpeed = (int)1000*cutSpeed/(piGreco*workDiameter);
    targetMin = spindleSpeed - toleranceSpeed*spindleSpeed;
    targetMax = spindleSpeed + toleranceSpeed*spindleSpeed;

    if(spindleSpeed <= maxSpindleSpeed){
      mylcd.Print_String("Spindle speed: " + String(spindleSpeed) + " rpm", 0,198);
      delay(1000);
      checkSpindleSpeed = 1;
    }else{
      mylcd.Print_String("The value of spindle speed is too high", 0, 198);
      mylcd.Print_String("Choose a lower value for the cutting speed", 0, 216);
      delay(1000);

      cuttingSpeed = selectCuttingSpeedB();
      cutSpeed = cuttingSpeed;
    }
  }while(checkSpindleSpeed == 0);
  return spindleSpeed;
}
```

If the computed spindle speed exceeds the upper bound, a value of the cutting velocity must be re-inserted: the code verifies the spindle speed with the new cutting value as soon as acceptable parameter has been set.

Once all the working parameters have been set and before starting the work, this value must be communicated to the Arduino board that handle the rotation of the spindle: this transfer will be performed later with suitable functions of the serial communication.

3.2.4 Definition of the feed along the Z-axis

The next step is to engage the feed relative to the movement along the Z-axis of the lathe. This parameter concerns the distance covered by the carriage and so of the tool when a complete turn of the piece is performed. The measurement unit is mm/rotation.

Code 21: Feed Z-axis selection

```
float feedZrough = 0.00; //[mm/rotation]

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Insert the feed rate along the Z-axis", 0, 18);
  mylcd.Print_String("with the keypad", 0, 36);
  mylcd.Print_String("between 0.35 and 0.65 mm/rotation", 0, 54);
  mylcd.Print_String("Press '#' to confirm the value", 0, 72);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
  mylcd.Print_String("re-insert a new one", 0, 108);
```

```
    feedZrough = feedRateZroughB(spindleSpeed);
    if(ok == 2){
      return;
    }
    delay(1000);
}
```

From the mechanical operation literature, the order of magnitude is around fractions of millimetres per rotation of the part. Different values can be chosen depending on the material to remove, the tool manufacturing and the quality of the moving components of the automatic machine. However, a general distinction is made for the typology of mechanical process to be machined, if a roughing or a finishing work cycle. The former involves a greater amount of material taken away regardless to the superficial roughness, so the feed can be chosen a bit higher than the one relative to the finishing stage, where the best superficial condition is the main requirement for the operation.

As before, this setting is imposed by the need of the user and so it is inserted through the keypad; a check is performed for its validation, specially it must be included in an acceptable range (code 22).

For the rough stages, the value must fall within 0,35 and 0,65 mm/rotation, while for the finishing part, where expected, a fixed feed is set, equal to 0,4.

The engagement of the feed reflects directly on the carriage system, of the stepper motor, the lead screw and the carriage itself. The speed of the longitudinal advance is acquired, according to the formula:

$$Longitudinal\ speed = V_a \left[\frac{mm}{min}\right] = a * n$$

Where “a” is the feed along the Z-axis and “n” the angular speed of the part. The velocity information affects the motor control, especially the period of the pulses sequence.

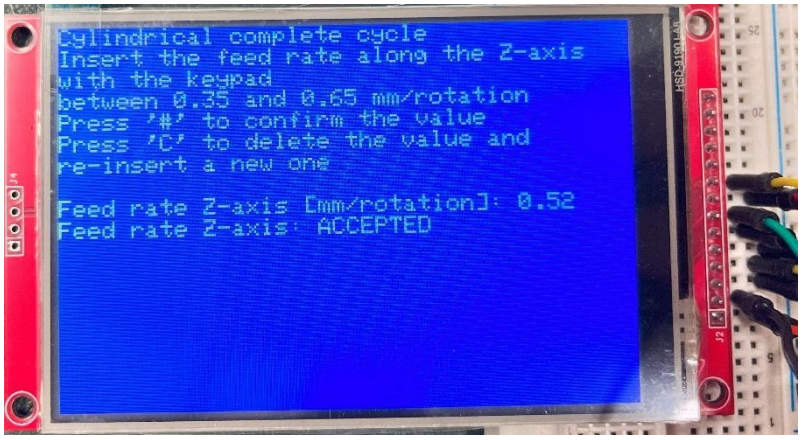


Figure 52: Selection of the feed rate of the Z-axis

Code 22: Feed Z-axis selection (detail)

```
float feedZ1 = 0.00;
bool checkFeedZ = 0.00;
String movementZ = "";

float feedRateZroughB(int rotationalSpeed){
```

```
  feedZ1 = 0;
  checkFeedZ = 0;
  movementZ = "";

  do{
    delay(50);
    char inCharKey = customKeypad.getKey();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      if(inCharKey != 'A' && inCharKey != 'B' && inCharKey != 'C' && inCharKey !=
'D' && inCharKey != '*' && inCharKey != '#'){
        //a number has been inserted
        movementZ += inCharKey;
        mylcd.Print_String("Feed rate Z-axis [mm/rotation]: " + movementZ, 0, 144);
        delay(50);
      }else if(inCharKey == '*'){ //'*' = '.'
        movementZ += '.';
        mylcd.Print_String("Feed rate Z-axis [mm/rotation]: " + movementZ, 0, 144);
        delay(50);
      }else if(inCharKey == 'C'){
        //clear the variable containing the parameter
        movementZ = "";
        mylcd.Fill_Screen(BLUE);
        mylcd.Print_String("Cylindrical complete cycle", 0, 0);
        mylcd.Print_String("Insert the feed rate along the Z-axis", 0, 18);
        mylcd.Print_String("with the keypad", 0, 36);
        mylcd.Print_String("between 0.35 and 0.65 mm/rotation", 0, 54);
        mylcd.Print_String("Press '#' to confirm the value", 0, 72);
        mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
        mylcd.Print_String("re-insert a new one", 0, 108);
        mylcd.Print_String("Feed rate Z-axis [mm/rotation]: ", 0, 144);
        delay(50);
      }else if(inCharKey == '#'){
        //all the digits of the diameter and the ok digit have been inserted
        feedZ1 = movementZ.toFloat();
        if(feedZ1 > 0 && (feedZ1 >= 0.35 && feedZ1 <=0.65)){
          mylcd.Print_String("Feed rate Z-axis: ACCEPTED", 0, 162);
          delay(1000);
          rotationalSpeedScrew = (int) (feedZ1*rotationalSpeed*pitch);
          checkFeedZ = 1;
        }else{
          mylcd.Print_String("Feed rate Z-axis not valid", 0, 162);
          mylcd.Print_String("Insert a new value", 0, 180);
          delay(1000);

          feedZ1 = 0;
          movementZ = "";
          mylcd.Fill_Screen(BLUE);
          mylcd.Print_String("Cylindrical complete cycle", 0, 0);
          mylcd.Print_String("Insert the feed rate along the Z-axis", 0, 18);
          mylcd.Print_String("with the keypad", 0, 36);
          mylcd.Print_String("between 0.35 and 0.65 mm/rotation", 0, 54);
          mylcd.Print_String("Press '#' to confirm the value", 0, 72);
          mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
          mylcd.Print_String("re-insert a new one", 0, 108);
        }
      }
    }else if(startState == LOW || secCheckState == HIGH){
      Emergency();
      ok = 2;
      return;
    }
  }while(checkFeedZ == 0);
  return feedZ1;
}
```

3.2.5 Acquisition of the working length

Last parameter that must be inserted before starting the turning operations regards the length on the piece to modify. It might not correspond to the whole length of the workpiece mounted on the chuck, due to the presence of the grabs themselves not allowing to the tool to work on the entire length or due to the need of the product.

Code 23: Acquisition working length

```
float workingLength = 0; //[mm]

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Insert the working length of the piece", 0, 18);
  mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 36);
  mylcd.Print_String("Press '#' to confirm the value", 0, 54);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
  mylcd.Print_String("re-insert a new one", 0, 90);

  workingLength = acquisitionWorkingLengthB();
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

It has been defined a maximum length of work, dictated by the physical sizes of the machine, set to 200 mm, and as usual a control has been added to the code to validate the insert.

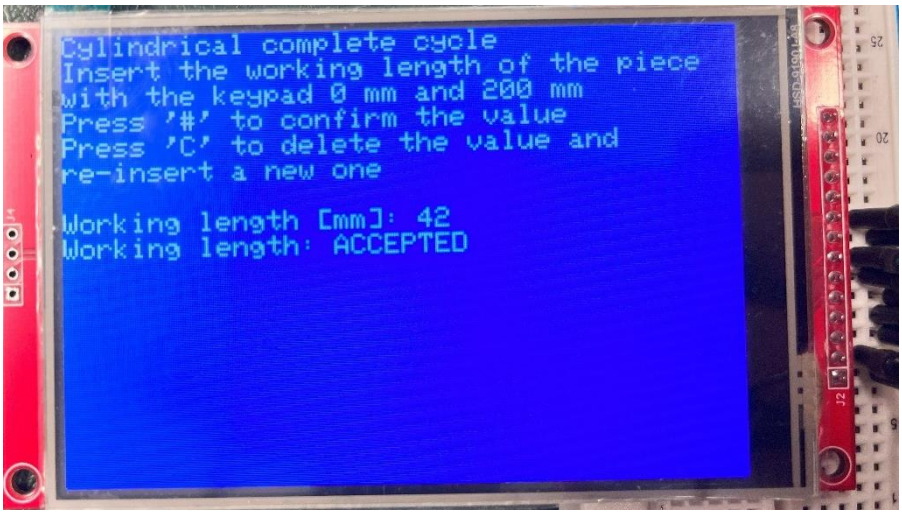


Figure 53: Selection of the working length

Code 24: Acquisition working length (detail)

```
float workingLength = 0;
bool checkWorkingLength = 0;
String workDistance = "";

float acquisitionWorkingLengthB(){
  workingLength = 0;
  checkWorkingLength = 0;
  workDistance = "";

  do{
    delay(50);
```

```
    char inCharKey = customKeypad.getKey();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      if(inCharKey != 'A' && inCharKey != 'B' && inCharKey != 'C' && inCharKey !=
'D' && inCharKey != '*' && inCharKey != '#'){
        //a number has been inserted
        workDistance += inCharKey;
        mylcd.Print_String("Working length [mm]: " + workDistance, 0, 126);
        delay(50);
      }else if(inCharKey == '*'){ //'*' = '.'
        workDistance += '.';
        mylcd.Print_String("Working length [mm]: " + workDistance, 0, 126);
        delay(50);
      }else if(inCharKey == 'C'){
        //clear the variable containing the parameter
        workDistance = "";
        mylcd.Fill_Screen(BLUE);
        mylcd.Print_String("Cylindrical complete cycle", 0, 0);
        mylcd.Print_String("Insert the working length of the piece", 0, 18);
        mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 36);
        mylcd.Print_String("Press '#' to confirm the value", 0, 54);
        mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
        mylcd.Print_String("re-insert a new one", 0, 90);
        mylcd.Print_String("Working length [mm]: ", 0, 126);
        delay(50);
      }else if(inCharKey == '#'){
        //all the digits of the diameter and the ok digit have been inserted
        workingLength = workDistance.toFloat();
        if(workingLength > 0 && workingLength <= maxLength){
          mylcd.Print_String("Working length: ACCEPTED", 0, 144);
          delay(2000);
          checkWorkingLength = 1;
        }else{
          mylcd.Print_String("Working length not valid", 0, 144);
          mylcd.Print_String("Insert a new value", 0, 162);
          delay(1000);

          workingLength = 0;
          workDistance = "";
          mylcd.Fill_Screen(BLUE);
          mylcd.Print_String("Cylindrical complete cycle", 0, 0);
          mylcd.Print_String("Insert the working length of the piece", 0, 18);
          mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 36);
          mylcd.Print_String("Press '#' to confirm the value", 0, 54);
          mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
          mylcd.Print_String("re-insert a new one", 0, 90);
        }
      }
    }else if(startState == LOW || secCheckState == HIGH){
      Emergency();
      ok = 2;
      return;
    }
  }while(checkWorkingLength == 0);
  return workingLength;
}
```

3.2.6 Communication spindle speed and start rotation

At this stage, all the required working parameters have been saved and the more operative passages can begin. Now the positioning phase can take place, and it is subdivided into a certain number of subroutines.

A crucial step regards the communication of the ideal angular speed of the spindle, and this data will represent the target signal for the feedback control scheme, as explained later in the dedicated chapter.

The Arduino tool used for this purpose is the serial transmission functionality, according to the UART protocol. As illustrated on the figure, for the communication two cables are required, each one is connected to the TX pin of one board on one side and to the RX pin of other board on the other side, and vice versa for the second cable. This cross-connection avails for the two directions of data flowing, if input or output for the Arduino board. For the speed exchange the Arduino containing the processes logic is the transmitter, while the Arduino board of management of the DC motor will be the receiver.

Among the three possible serial channels, for the project only one has been activated.

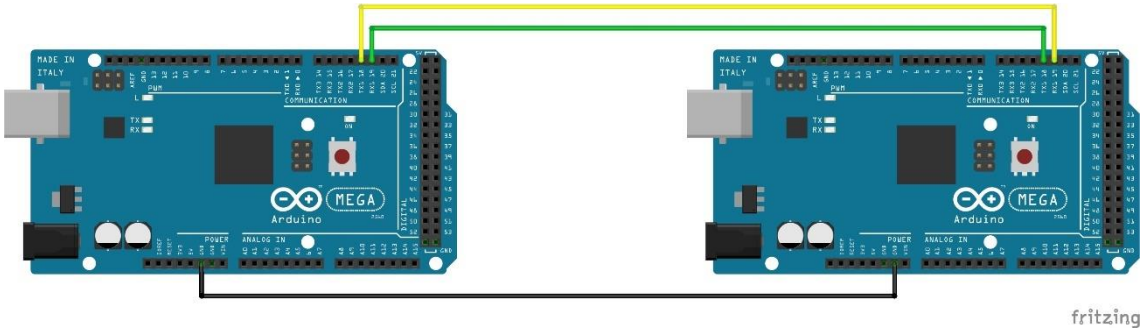


Figure 54: Serial communication harness

The spindle control program on the other Arduino must be inside a specific function waiting for the data reception: this is executed as soon as the enabling signal at the beginning of the process is sent to the second Arduino board, so it is prepared to work correctly.

Then, the “Serial3.println” function executes the real exchange of the integer variable (code 25). The succeed of the operation is confirmed by a sequence of data exchange: if all these passages are executed correctly and the safety conditions are met, a confirm message is visualized on the HMI display and the logic can go through.

Code 25: Speed communication

```
int enb_spd = 0;
bool receiveOK = 0;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  delay(500);

  sendSpindleSpeed();
  if(ok == 2){
    return;
  }
}

void sendSpindleSpeed(){
  Serial3.println(spindleSpeed);
  do{
    enb_spd = receiveEnableSpeed();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
```

```
  }while(startState == HIGH && secCheckState == LOW && enb_spd == 0);

  if(startState == LOW || secCheckState == HIGH){
    Emergency();
    ok = 2;
    return;
  }
  mylcd.Print_String("Transmission of the spindle", 0, 18);
  mylcd.Print_String("speed complete", 0, 36);
  delay(500);
}

int receiveEnableSpeed(){
  enb_spd = 0;
  receiveOK = 0;
  do{
    if(Serial3.available()>0){
      String receivedString = Serial3.readStringUntil('\n');
      enb_spd = receivedString.toInt();
      if(enb_spd == 1){
        receiveOK = 1;
      }
    }
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
  }while(startState == HIGH && secCheckState == LOW && receiveOK == 0);
  delay(1000);
  return enb_spd;
}
```

Once the reference speed has been set, the real start of the process must come from the operator: before the enclosure of the security window of the spindle (meaning that the workpiece has been mounted and finally secured with the suitable grabs), and as final stage an enable command is pressed (code 26).

Code 26: Check on the security window position

```
#define secCheckPin 36
bool secCheckState = 0;

bool okEnable = 0;
int enb_rot = 0;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Transmission of the spindle", 0, 18);
  mylcd.Print_String("speed complete", 0, 36);
  mylcd.Print_String("Close the window safely", 0, 72);

  do{
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    delay(100);
  }while(startState == HIGH && secCheckState != HIGH);

  if(startState == LOW || secCheckState == LOW){
    Emergency();
    ok = 2;
    return;
  }
}

  okEnable = 0;
  mylcd.Print_String("Press # to enable", 0, 108);
```

```
  mylcd.Print_String("the beginning of the process", 0, 126);

  do{
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    delay(100);

    varEnable = '0';
    varEnable = customKeypad.getKey();
    if(varEnable && varEnable == '#'){
      sendEnableRotation();
      delay(1000);
      okEnable = 1;
    }
  }while(okEnable == 0 && startState == HIGH && secCheckState == HIGH);

  if(startState == LOW || secCheckState == LOW){
    Emergency();
    ok = 2;
    return;
  }
  delay(1000);
}

void sendEnableRotation(){
  enb_rot = 1;
  Serial3.println(enb_rot);
}
```

A modification has been brought to the stop condition coming from the check on the spindle security window: from this stage and so on, if the window opens suddenly (from HIGH to LOW logic level), the emergency function is called and the program stops its execution (differently from the previous condition, where the window must be kept open to go through the program without interruptions).

Once the rotation has started, the main Arduino program jumps inside a cyclic structure waiting for the settling of the steady state condition, that is a stable rotational velocity around the target value. The confirm of this achievement occurs with another serial communication: this time the exchanged and incoming data is the real angular speed. If it is included within the tolerance range of ±2% the positioning phase can start.

Once this becomes available, the data are saved primarily into a string variable and subsequently convert into an integer for its validation.

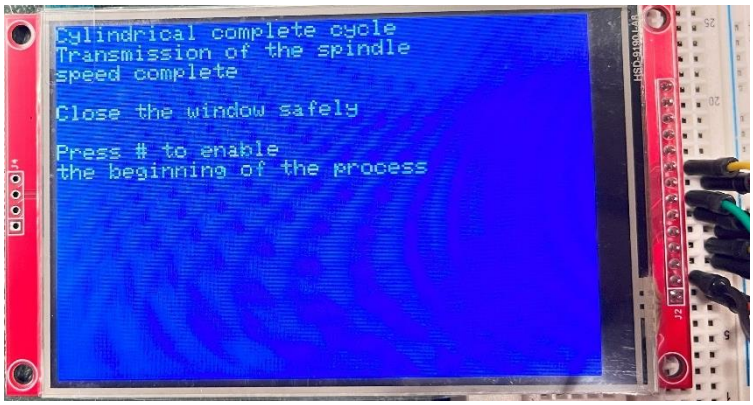


Figure 55: Operations to start the work cycle

Code 27: Real speed reception

```
bool okSpeed = 0;
int realSpindleSpeed = 0;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Transmission of the spindle", 0, 18);
  mylcd.Print_String("speed complete", 0, 36);
  mylcd.Print_String("Close the window safely", 0, 72);
  mylcd.Print_String("Press # to enable", 0, 108);
  mylcd.Print_String("the beginning of the process", 0, 126);

  realSpindleSpeed = 0;
  do{
    realSpindleSpeed = receiveSpindleSpeed();
  }while(startState == HIGH && secCheckState == HIGH && realSpindleSpeed == 0);

  if(startState == LOW || secCheckState == LOW){
    Emergency();
    ok = 2;
    return;
  }
  delay(1000);
}

int receiveSpindleSpeed(){
  okSpeed = 0;
  realSpindleSpeed = 0;
  do{
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(Serial3.available()>0){
      String receivedControlSpeed = Serial3.readStringUntil('\n');
      realSpindleSpeed = receivedControlSpeed.toInt();

      if((realSpindleSpeed >= targetMin) && (realSpindleSpeed <= targetMax)){
        okSpeed = 1;
        mylcd.Print_String("Real spindle speed: "+ String(realSpindleSpeed),0,162);
        delay(500);
      }
    }
  }while(okSpeed == 0 && startState == HIGH && secCheckState == HIGH);
  return realSpindleSpeed;
}
```

3.2.7 Positioning phase

The motion of the tool can be subdivided into three phases: the positioning, the work and the return. The first and the third ones are common to all the processes available on the program, instead the work cycle is specific of the desired mechanical operation.

Code 28: Positioning phase

```
int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  positioningProcessB(workingLength, feedXrough);
  if(ok == 2){
    return;
  }
  delay(1000);
```

```
}
```

In turn the positioning stage includes the measurement of the distance for the approach of the tool, and the precise motion of the tool itself according to the work parameters.

The carriage during the non-work time is placed in a fixed and retracted position, on the full right of the guides of the machine bed.

The measure of the distance regards the longitudinal length between the rest position and one edge of the mounted piece, and it is performed by the ultrasonic sensor.

The origin of the machine reference frame (Z and X axes) must coincide with the rest position of the tool and the zero-value of the sensor.

As explained in the dedicated paragraph, the sensor is able to evaluate this distance by transmitting a series of soundwaves and computing the time elapsed at the moment of the receiving: knowing the sound speed, the length is calculated.

For a better estimation of the distance, a series of measurements is computed and then the average value is saved.

The actual value is decreased by an extra-length: the result is the Z-axis coordinate where the tool must stop to allow the X-axis positioning without touching the part to avoid malfunctions and damages of the product, before the real mechanical work.

Code 29: Measure of the positioning distance

```
bool checkOverallDistance = 0;
#define trigPin 22 //generates the soundwave to send to the object
#define echoPin 23 //listen to the soundwave coming back from the object
float duration = 0;
float distance = 0;
float measures[101];
float sum = 0;
float averageDistance = 0; //[mm]

const float extra_len = 5; //extra-distance [mm]
float positioning = 0; //[mm]
float tot_distance = 0;

void positioningProcessB(float cycleLength, float feedAlongX){
  //function to position the tool near the piece to work/to the starting position
  do{
    checkOverallDistance = 0;

    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Cylindrical complete cycle", 0, 0);
    mylcd.Print_String("Measure of the distance:", 0, 18);

    //Acquire 100 measures and perform the average
    for(int i=0; i<101; i++){
      digitalWrite(trigPin, LOW);
      delayMicroseconds(2);
      digitalWrite(trigPin, HIGH);
      delayMicroseconds(10);
      duration = pulseIn(echoPin, HIGH);
      distance = (duration*0.343)/2;
      measures[i] = distance;
      delay(10);
    }
    sum = 0;
```

```
    averageDistance = 0;
    for(int i=1; i<101; i++){
      sum = sum + measures[i];
    }
    averageDistance = sum/100;
    positioning = averageDistance - extra_len;
    tot_distance = averageDistance + cycleLength;
    if(tot_distance <= maxLength){
      mylcd.Print_String("Measure of the distance: " + String(averageDistance) + "
mm", 0, 18);
      delay(1000);
      checkOverallDistance = 1;
    }else{
      mylcd.Fill_Screen(BLUE);
      mylcd.Print_String("Cylindrical complete cycle", 0, 0);
      mylcd.Print_String("ERROR in the positioning!", 0, 18);
      delay(1000);
  }while(checkOverallDistance == 0);
}
```

Before moving the carriage, a check of the validity of the measure is required: the sum of the work length and the former measure must be smaller than the maximum length of the machine bed of the admissible movements. If the output of this control is negative, a message error is displayed, and the process is immediately halted, due to the need of changing the work part.

On the contrary, the positioning can start: firstly, the longitudinal coordinate is reached, then the transversal feed is engaged.

Since the stepper motor performs a rotation by applying a sequence of a certain number of pulses, it is acquired by multiplying the distance to cover for the pulses required to do a 360° revolution. The result is divided by the pitch of the lead screw and in this case, it is equal to 1 mm.

After the initialization of the required variables to track the motion and the definition of the rotation direction, the loop structure is executed.

Since during this time the tool does not work on the piece, the angular speed of the stepper motor has been assumed as a default value, of 200 rpm and the corresponding half-period of the square wave is inserted into the relative code commands.

Important variables are the “comeback Z” and then the “comeback X” ones: they store all the pulses required for both the motors to cover the positioning and the working lengths.

The float variable “distance Z” instead is useful to display the progressive movement of the carriage: every 1 mm covered, the coordinate Z is updated and shown on the first row of an LCD display (Fig. 56).

Code 30: Positioning along the Z-axis

```
float pitch = 1.00; //pitch of the lead screw [mm]
long pulsesZ = 0;
int NpulsesRevolutionZ = 0;
float distanceZ = 0;
volatile long comebackVarZ = 0;
long countZ = 0;

void setup(){
  NpulsesRevolutionZ = 800;
```

```
  T_screw = 60*1e6;
  T_screw = T_screw/2;
  T_screw = T_screw/200;
  T_screw = T_screw/NpulsesRevolutionZ;
  halfT_screw = (int)T_screw;
}

void positioningProcessB(float cycleLength, float feedAlongX){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Measure of the distance: "+ String(averageDistance) + " mm",
0, 18);

  mylcd.Print_String("Positioning of the tool", 0, 36);
  delay(500);

  //move along the longitudinal axis (Z-axis)
  pulsesZ = NpulsesRevolutionZ*positioning/pitch;

  lcd.clear();
  mylcd.Print_String("Moving along the Z-axis:", 0, 54);
  distanceZ = 0.00;
  comebackVarZ = 0;
  countZ = 0;
  digitalWrite(dirPin, HIGH);
  for(int i=0; i<pulsesZ; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin, HIGH);
    delayMicroseconds(halfT_screw);
    digitalWrite(stepPin, LOW);
    delayMicroseconds(halfT_screw);
    comebackVarZ++;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceZ = comebackVarZ/NpulsesRevolutionZ;
      countZ = 0;
      lcd.home();
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  distanceZ = positioning;
  mylcd.Print_String("Z position: " + String(distanceZ) + "/" + String(positioning)
+ " mm", 0, 72);
  lcd.home();
  lcd.print("Z value: " + String(distanceZ) + " mm");
  delay(500);
}
```

The cross slide, which is placed on the carriage holding the tool, moves following a similar procedure: before the positioning distance must be defined. It depends on the distance between the rest position and the X-coordinate corresponding to the initial diameter of the piece. It is the sum of two contributions: the first is a fixed amount, which is the length between the non-work pose and the maximum diameter admissible on the chuck, and the second is a changeable amount rising from the difference of the previous diameter and the actual initial diameter.

Similarly to before, the float variable “distance X” is useful to display the progressive movement of the cross-slide: every 1 mm covered, the coordinate X is updated and shown on the second

row of the LCD display (Fig. 56). This information about the position of the tool along the motion will be maintained also for the following work phases.

Figure 56: Visualization of the coordinates of the tool

Code 31: Positioning along the X-axis

```
volatile long comebackVarX = 0;
float positionXaxis = 0;
float xHome = 5;
long pulsesPosX = 0;
float maximumDiameter = 30.00; //maximum acceptable working diameter of the piece
[mm]

const int gearRatio = 64; //gear ratio of the stepper motor in the X-axis
const int angleX = 45; //angle covered by the X-axis motor when a cycle of
activation of the 4 phases
float pitch = 1.00; //pitch of the screw of the Z-axis and X-axis [mm]
int NpulsesRevolutionX = 512;

float distanceX = 0;
long countX = 0;

void positioningProcessB(float cycleLength, float feedAlongX){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Measure of the distance: "+ String(averageDistance) + " mm",
0, 18);
  mylcd.Print_String("Positioning of the tool", 0, 36);
  mylcd.Print_String("Moving along the Z-axis:", 0, 54);
  mylcd.Print_String("Z position: " + String(distanceZ) + "/" + String(positioning)
+ " mm", 0, 72);
  delay(500);

  //positioning of the tool along the X-axis: from the home position to the initial
diameter
  mylcd.Print_String("Moving along the X-axis:", 0, 108);

  comebackVarX = 0;
  positionXaxis = xHome + (maximumDiameter - startDiameter)/2;
  pulsesPosX = positionXaxis*gearRatio*360/(pitch*angleX);

  delayMotor = 2;
  distanceX = 0.00;
  countX = 0;
  for(int i=0; i<pulsesPosX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    Forward();
    comebackVarX++;
    countX++;
```

```
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }
  mylcd.Print_String("X position: " + String(positionXaxis) + " mm", 0, 126);
  lcd.setCursor(0,1);
  lcd.print("X value: " + String(distanceX)  + " mm");
  delay(500);
}
```

Finally, the tool must reach the precise position along the X-axis according to the desired feed X. Since a maximum value of depth for one work pass has been defined, to preserve the integrity of the shape of the tool, two possibilities can arise. If the engaged feed is smaller than this maximum quantity, all the excess material is removed in only one pass; otherwise, more repetitions are needed. For the first pass, the feed engaged is set to 0,5 mm.

Code 32: Positioning according to the feed X value

```
float maxFeedX = 0.5;
long internVarX = 0;
long pulsesX = 0;

void positioningProcessB(float cycleLength, float feedAlongX){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Measure of the distance: "+ String(averageDistance) + " mm",
0, 18);
  mylcd.Print_String("Positioning of the tool", 0, 36);
  mylcd.Print_String("Moving along the Z-axis:", 0, 54);
  mylcd.Print_String("Z position: " + String(distanceZ) + "/" + String(positioning)
+ " mm", 0, 72);
  mylcd.Print_String("Moving along the X-axis:", 0, 108);
  mylcd.Print_String("X position: " + String(positionXaxis) + " mm", 0, 126);
  delay(500);

  //move along the trasversal axis (X-axis) to reach the desired feed X
  mylcd.Print_String("Position along the X-axis of the feed:", 0, 162);
  if(feedAlongX <= maxFeedX){
    pulsesX = feedAlongX*gearRatio*360/(pitch*angleX);

    internVarX = 0;
    delayMotor = 2;
    for(int i=0; i<pulsesX; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      Forward();
      comebackVarX++;
      internVarX++;
      countX++;
      if(countX == NpulsesRevolutionX){
        distanceX = comebackVarX/NpulsesRevolutionX;
        countX = 0;
        lcd.setCursor(0,1);
        lcd.print("X value: " + String(distanceX) + " mm");
      }
    }
```

```
    distanceX = feedAlongX;
    mylcd.Print_String("X position: " + String(distanceX) + " mm", 0, 180);
    lcd.setCursor(0,1);
    lcd.print("X value: " + String(distanceX)  + " mm");
    delay(1000);
  }else if(feedAlongX > maxFeedX){
    pulsesX = maxFeedX*gearRatio*360/(pitch*angleX);
    internVarX = 0;
    delayMotor = 2;
    for(int i=0; i<pulsesX; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      Forward();
      comebackVarX++;
      internVarX++;
      countX++;
      if(countX == NpulsesRevolutionX){
        distanceX = comebackVarX/NpulsesRevolutionX;
        countX = 0;
        lcd.setCursor(0,1);
        lcd.print("X value: " + String(distanceX) + "mm");
      }
    }
    distanceX = maxFeedX;
    mylcd.Print_String("X position: " + String(distanceX) + " mm", 0, 180);
    lcd.setCursor(0,1);
    lcd.print("X value: " + String(distanceX) + " mm");
    delay(1000);
  }
}
```

The relative stepper motor is commanded as before and finally the tool is placed in the right position to start the real work cycle.

3.2.8 Stop signal, return phase and emergency stop

Once the work cycle is completed successfully, a stop signal is generated: it is transmitted through the serial communication to the other Arduino control board to interrupt the rotation of the piece.

Code 33: Stop signal communication

```
int enb_stop = 0;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  //send to the other Arduino the command to stop the rotation
  sendStop();
  delay(1000);
}

void sendStop(){
  enb_stop = 1;
  Serial3.println(enb_stop);
}
```

Now the carriage system must move back to the rest position, along both the motion axes: the variables “comeback X” and “comeback Z” come in play. They store all the pulses performed by the stepper motors during all the positioning and work phases, and they are crucial to establish the correct conditions for the next mechanical operation.

Code 34: Back motion function

```
int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  backHome(comebackVarZ,comebackVarX);
  delay(1000);
}

void backHome(long backZ, long backX){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Moving back the tool", 0, 0);
  delay(1000);

  //before back movement along the X-axis
  mylcd.Print_String("along the X-axis", 0, 18);
  delayMotor = 2;
  for(int i=0; i<backX; i++){
    Backward();
    comebackVarX--;
    countX++;
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }

  //after back movement along the Z-axis
  mylcd.Print_String("along the Z-axis", 0, 36);
  digitalWrite(dirPin,LOW);
  for(int i=0; i<backZ; i++) {
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(halfT_screw);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(halfT_screw);
    comebackVarZ--;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceZ = comebackVarZ/NpulsesRevolutionZ;
      countZ = 0;
      lcd.home(0,1);
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  delay(1000);

  mylcd.Print_String("Process completed correctly", 0, 72);
  delay(1000);
}
```

Before the cross slide is actuated, subsequently the carriage is moved backward longitudinally.

The final stage of the process is the opening of the security window of the spindle: now the operator can lift it up safely and disassemble the product.

Figure 57: Final operations

Code 35: Final stages

```
int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  //check the position of the security window
  mylcd.Print_String("Open the window safely", 0, 108);
  lcd.clear();  lcd.print("Open the
window");  lcd.setCursor(0,1);  lcd.print("safely");
  do{
    secCheckState = digitalRead(secCheckPin);
    delay(200);
  }while(secCheckState != LOW); //startState != HIGH ||

  mylcd.Print_String("Ready for a new process", 0, 144);
  delay(2000);

  ok = 1;
  return ok;
}

void loop(){
  startState = digitalRead(startPin); //acquisition of the signal from the start
connection
  secCheckState = digitalRead(secCheckPin);

  if(startState == HIGH && secCheckState == LOW){
    do{
      ok = 0;
      delay(50);
      startState = digitalRead(startPin);
      varMode = customKeypad.getKey();

      if(varMode && startState == HIGH){ //this is necessary because the varMode
variable contains something, otherwise in the switch function the default branch is
always selected
        switch(varMode){
          case 'A': lcd.clear();
            ok = completeProcess();
            lcd.clear();delay(1000);
            mylcd.Fill_Screen(BLUE);
            lcd.clear();
          break;
        }
      }else if(startState == LOW){
        Emergency();
```

```
        ok = 2;
      }
    }while(ok == 0);
  }else if(startState == LOW || secCheckState == HIGH){
    lcd.clear();
    mylcd.Fill_Screen(BLUE);
  }
}
```

The code shows the correct execution also on the HMI screen and restores the initial conditions, where the user can start again the machine or prepare a new mechanical work process.

A safety command present in all the automatic and manual operational machines is the possibility to abort the work cycle any time in case of any misbehaviour.

This functionality is expected and developed throughout the execution of all the processes. It is triggered by the reading of the start signal, especially if it turns off during any phase of the work.

Another signal can interrupt the process, and it comes from the reading of the security window position: if the code detects its closed position during the setup procedure or its open position during the positioning and job phases, the execution must halt immediately. In the first case, the effect is the interruption of the parameters acquisition to go back to the start condition, while for the second scenario the stop of the rotation of the piece and the movement of the tool occurs. The code 36 example contains both the cases, where the interruption occurs inside a parameter setting or during a work execution.

Code 36: Emergency function activation

```
int completeProcess(){
  startDiameter = acquisitionInitialDiameterB();
  if(ok == 2){
    return;
  }
}

float acquisitionInitialDiameterB(){
  do{
    delay(50);
    char inCharKey = customKeypad.getKey();
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(inCharKey && startState == HIGH && secCheckState == LOW){
      ...
    }else if(startState == LOW || secCheckState == HIGH){
      Emergency();
      ok = 2;
      return;
    }
  }while(checkDiameter == 0);
  return startDiameter;
}

long workingProcess1Cycle(float length, int screwSpeed){
  pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;
  digitalWrite(dirPin,HIGH);
  for(int i=0; i<pulses; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
```

```
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(periodSpeed);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(periodSpeed);
    comebackVarZ++;
  }
  return comebackVarZ;
}
```

Inside the emergency function, the retraction of the tool can occur only if the tool has already moved from its rest position, and it is enabled by pressing a specific key.

Figure 58: Key to enable the retraction of the tool and emergency stop operations

Code 37: Emergency function (detail)

```
#define emergencyPin 19
bool emergencyState = false;

char varBack;
bool varBackstate = 0;

void setup(){
  pinMode(emergencyPin, OUTPUT);
  emergencyState = false;
}

void Emergency(){
  //EMERGENCY STOP AND THE TOOL COMES BACK TO THE HOME POSITION
  emergencyState = !emergencyState; //the state becomes true
  digitalWrite(emergencyPin, emergencyState); //send the info to the other Arduino

  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Emergency stop", 0, 0);
  delay(1000);

  if(comebackVarX != 0 || comebackVarZ != 0){
    mylcd.Print_String("Press the 'D' command", 0, 36);
    mylcd.Print_String("to move back the tool", 0, 54);

    varBackstate = 0;
    do{
      varBack = customKeypad.getKey();
      delay(50);

      if(varBack && varBack == 'D'){
        varBackstate = 1;
```

```
      }
    }while(varBackstate == 0);

    mylcd.Print_String("Moving back the tool", 0, 90);

    //acquire the position of the motors and move back the tool
    for(int i=0; i<comebackVarX; i++){
      Backward();
    }

    digitalWrite(dirPin,LOW);
    for(int j=0; j<comebackVarZ; j++){
      digitalWrite(stepPin,HIGH);
      delayMicroseconds(halfT_screw);
      digitalWrite(stepPin,LOW);
      delayMicroseconds(halfT_screw);
    }

    comebackVarZ = 0;
    comebackVarX = 0;

    mylcd.Print_String("Open the security window", 0, 126);
    do{
      secCheckState = digitalRead(secCheckPin);
      delay(100);
    }while(secCheckState == HIGH);
  }else{
    comebackVarZ = 0;
    comebackVarX = 0;

    mylcd.Print_String("Open the security window", 0, 36);
    do{
      secCheckState = digitalRead(secCheckPin);
      delay(100);
    }while(secCheckState == HIGH);
  }

  emergencyState = !emergencyState; //the state becomes false
  digitalWrite(emergencyPin,emergencyState);
  mylcd.Fill_Screen(BLUE);
}
```

The exit condition from the emergency function is the check on the position of the window: it must be open as suggest by the HMI message.

The following sections describe the precise work cycles machined, the cylindrical mono-dimensional and multidimensional cylindrical turnings and the conical process. The explanation regards the subdivision of the job into the roughing and the finishing stages and other particular functions.

3.3 Cylindrical turning: complete working cycle

The first and simpler operation that can be implemented on a lathe machine is the cylindrical turning with a constant diameter all along the work length. An example of the final shape is the one illustrated in the Fig. 59.

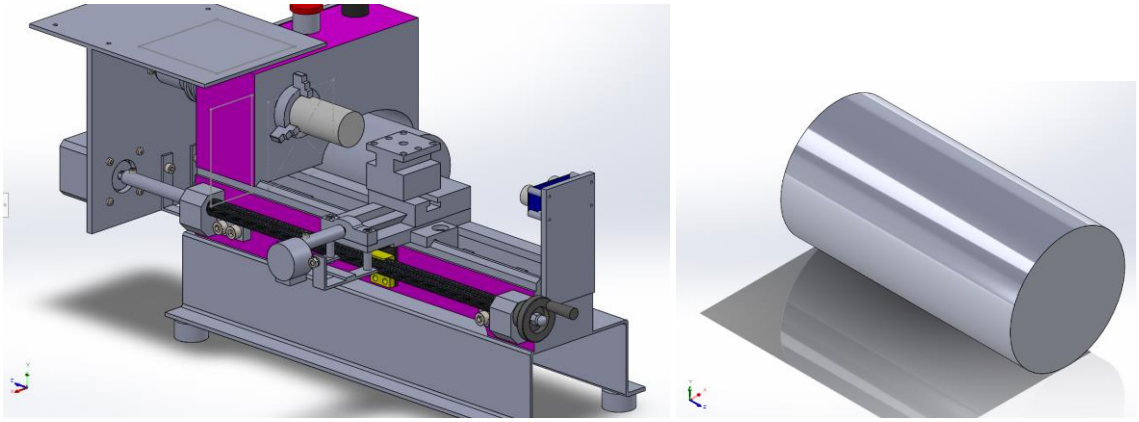


Figure 59: 3D models of the cylindrical turning product

This turning option can be selected by the user pressing the 'A' code on the HMI keyboard as specified by the message on the screen (Fig. 60).

Figure 60: Key to select the complete cylindrical turning

Code 38: Complete cylindrical turning selection

```
void loop(){
  startState = digitalRead(startPin); //acquisition of the signal from the start
connection
  secCheckState = digitalRead(secCheckPin);

  if(startState == HIGH && secCheckState == LOW){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Choose a working process ", 0, 0);
    mylcd.Print_String("from the following list:", 0, 18);
```

```
    mylcd.Print_String("> A: complete cylindrical turning with", 0, 36);
    mylcd.Print_String("roughing + finishing stages", 0, 54);
    mylcd.Print_String("> B: cylindrical turning for a", 0, 72);
    mylcd.Print_String("multidiameter component with", 0, 90);
    mylcd.Print_String("roughing + finishing stages", 0, 108);
    mylcd.Print_String("> C: conical turning with", 0, 126);
    mylcd.Print_String("roughing + finishing stages", 0, 144);
    mylcd.Print_String("> D: cylindrical roughing turning", 0, 162);
    mylcd.Print_String("Or * to exit", 0, 180);

    do{
      ok = 0;
      delay(50);
      startState = digitalRead(startPin);
      varMode = customKeypad.getKey();

      if(varMode && startState == HIGH){
        switch(varMode){
          case 'A': lcd.clear();
            ok = completeProcess();
            delay(1000);
            mylcd.Fill_Screen(BLUE);
            lcd.clear();
          break;
        }
      }else if(startState == LOW){
        Emergency();
        ok = 2;
      }
    }while(ok == 0);
  }else if(startState == LOW || secCheckState == HIGH){ //OFF state
    lcd.clear();
    mylcd.Fill_Screen(BLUE);
  }
}
```

To obtain the final product, the whole process is subdivided into 2 stages: the roughing and the finishing. The former executes the majority removal of the material, to obtain a final shape adhered to the desired final diameter and regardless of the superficial roughness. The latter stage, instead, performs a unique pass with a low feed to remove a very limited amount of material, just to even the profile and reduce the roughness. The second work is carried out with a decreased and fixed feed along the Z-axis.

To execute this operation, the setup stage must acquire the initial and final dimensions of the piece, its work length and all the machine parameters. Depending on the sizes and so the amount of material to remove, the number of passes is defined starting from the feed along the X-axis (code 39).

Code 39: Computation of the feed X value

```
float feedXint = 0;
float feedXfinish = 0.010;
float maxFeedX = 0.5;
int numberCycles = 0;
float ver = 0;

int completeProcess(){
  feedXrough = checkFeedXcomplete(startDiameter,finalDiameter);
  delay(1000);
}

float checkFeedXcomplete(float initialDiameter, float workDiameter){
```

```
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);
  mylcd.Print_String("Initial diameter: " + String(initialDiameter) + " mm", 0,18);
  mylcd.Print_String("Final diameter: " + String(workDiameter) + " mm", 0, 36);

  feedXint = (initialDiameter - workDiameter)/2;
  feedXint = feedXint - feedXfinish;
  if(feedXint <= maxFeedX){
    numberCycles = 1;
    mylcd.Print_String(String(numberCycles) + " cycle needed", 0, 72);
    delay(2000);
  }else{
    numberCycles = (int)(feedXint/maxFeedX);
    ver = feedXint - numberCycles*maxFeedX;
    if(ver != 0){
      numberCycles++;
    }
    mylcd.Print_String(String(numberCycles) + " cycles needed", 0, 72);
    delay(2000);
  }
  return feedXint;
}
```

At code level, the distinction in the two stages is implemented by decreasing the total feed X by the finishing one, that is equal to 10 µm (acquired from mechanical operation manuals).

The practical evaluation of the cycles number depends on the number previously computed and by its comparison with the maximum feed allowed for one pass.

After the setting of all the parameters required, the communication with the other control board and the positioning of the carriage system, the work cycle can begin, before with the roughing phase and then with the finishing one.

3.3.1 Working phase: roughing stage

Based on the previous evaluation, firstly the program chooses one of the two alternatives based on the number of cycles needed (code 40).

Code 40: Definition of the work phases

```
int completeProcess(){
  if(numberCycles == 1){
    comebackVarZ = workingProcess1Cycle(workingLength, rotationalSpeedScrew);
  }else if(numberCycles > 1){
    comebackVarZ = workingProcessMoreCycles(workingLength, rotationalSpeedScrew,
feedXrough);
  }
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

When only one pass must be manufactured, the cycle is divided into the real mechanical job and then the return phase for firstly the cross slide and subsequently for the carriage up to the positioning pose (code 41).

The first evaluation is the computation of the number of pulses needed by the Z-axis stepper motor: it arises from the multiplication of the steps per rotation (800) and the length to cover,

that is the sum of the work length and the extra-length before the part (the result is then divided by the pitch of the lead screw).

Inside the initialization of all the needed variables to keep track of movements, the half period for the stepper motor of the Z-axis is computed based on the feed along the Z-axis and so the angular speed of the lead screw.

The core part of the work cycle is now commanded with the loop structure useful to send the right pulses sequence to the motor.

Inside the loop, the variables storing the number of steps are updated for each iteration: one variable is the global one (positioning and work phases) and the other inner of the work cycle.

Code 41: Z-axis work cycle

```
long pulses = 0;
float distanceCovered = 0.00;
long internVarZ = 0;

float halfperiodSpeed = 0;
int periodSpeed = 0;

long workingProcess1Cycle(float length, int screwSpeed){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working roughing cycle:", 0, 0);
  delay(500);

  pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;

  distanceCovered = 0.00;
  internVarZ = 0;

  halfperiodSpeed = 60*1e6;
  halfperiodSpeed = halfperiodSpeed/(2*NpulsesRevolutionZ);
  halfperiodSpeed = halfperiodSpeed/screwSpeed;
  periodSpeed = (int) halfperiodSpeed;

  digitalWrite(dirPin,HIGH);
  for(int i=0; i<pulses; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(periodSpeed);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(periodSpeed);
    comebackVarZ++;
    internVarZ++;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceCovered++;
      distanceZ = comebackVarZ/NpulsesRevolutionZ;
      countZ = 0;
      lcd.home();
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  distanceCovered = length + extra_len;
```

```
  distanceZ = positioning + distanceCovered;
  mylcd.Print_String("Worklength: " + String(length) + "/" + String(length) + "
mm", 0, 18);
  lcd.home();
  lcd.print("Z value: " + String(distanceZ) + " mm");
}
```

On the HMI display the coordinates of the tool are visualized thanks to the updating of a dedicated counter and the Z value updates accordingly throughout the process.

Once the forward motion of the tool terminates, both the stepper motors must retreat. Immediately the back-motion along the X-axis (code 42) occurs and then along the Z-axis (code 43): the tool moves away from the workpiece without touching it anymore, because its rotation continues and does not stop. The two internal variables ("internVarX" and "internVarZ") used to store the pulses of the work cycle are used to perform these back motions.

Code 42: Transversal back-motion

```
long workingProcess1Cycle(float length, int screwSpeed){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working roughing cycle:", 0, 0);
  mylcd.Print_String("Worklength: " + String(length) + "/" + String(length) + "
mm", 0, 18);

  mylcd.Print_String("Return phase", 0, 54);
  mylcd.Print_String("Move back along X-axis", 0, 72);
  delayMotor = 2;
  for(int i=0; i<internVarX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    Backward();
    comebackVarX--;
    countX++;
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }
  delay(500);
}
```

Code 43: Longitudinal back-motion

```
long workingProcess1Cycle(float length, int screwSpeed){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working roughing cycle:", 0, 0);
  mylcd.Print_String("Worklength: " + String(length) + "/" + String(length) + "
mm", 0, 18);
  mylcd.Print_String("Return phase", 0, 54);
  mylcd.Print_String("Move back along X-axis", 0, 72);

  mylcd.Print_String("Move back along Z-axis", 0, 90);
  digitalWrite(dirPin,LOW);
  for(int i=0; i<internVarZ; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
```

```
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(halfT_screw);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(halfT_screw);
    comebackVarZ--;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceCovered++;
      distanceZ = comebackVarZ/NpulsesRevolutionZ;
      countZ = 0;
      lcd.home();
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  return comebackVarZ;
}
```

During the movement the steps variables are updated accordingly.

When the number of passes is greater than 1, an additional structure is required, and it consists of a loop function to keep track of the repetitions (code 44).

The sequence of motions of the tool is the same as before: forward move along Z-axis, backward move along X-axis and then Z-axis. After the last movement, a check must be performed. The tool must place according to the number of passes to execute: if the next pass is not the last, the feed X value will be again the maximum (multiplied by the number of passes already machined), on the contrary, if the repetition is the last one, the feed X covered will be the overall value to obtain the final size of the product.

Code 44: Multicycle work phase

```
long workingProcessMoreCycles(float length, int screwSpeed, float feedXaxis){
  pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;

  for(int Ncycle=0; Ncycle<numberCycles; Ncycle++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(Ncycle+1), 0, 18);
    mylcd.Print_String("Z-position: " + String(distanceCovered) + " mm", 0, 36);
    mylcd.Print_String("Return phase", 0, 72);
    mylcd.Print_String("Move back along X-axis", 0, 90);
    mylcd.Print_String("Move back along Z-axis", 0, 108);

    //re-positioning along the X-axis
    nC = Ncycle+2;
    if(nC < numberCycles){
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 144);
      pulsesX = maxFeedX*NpulsesRevolutionX*(Ncycle+2);
      internVarX = pulsesX;
      delayMotor = 2;
      for(int i=0; i<pulsesX; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
```

```
        }
        Forward();
        comebackVarX++;
        countX++;
        if(countX == NpulsesRevolutionX){
          distanceX = comebackVarX/NpulsesRevolutionX;
          countX = 0;
          lcd.setCursor(0,1);
          lcd.print("X value: " + String(distanceX)  + " mm");
        }
      }
      mylcd.Print_String("X-position: " + String(maxFeedX) + " mm", 0, 162);
      delay(1000);
    }else if(nC == numberCycles){
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 144);
      pulsesX = (int) NpulsesRevolutionX*feedXaxis;
      internVarX = pulsesX;
      delayMotor = 2;
      for(int i=0; i<internVarX; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
        }
        Forward();
        comebackVarX++;
        countX++;
        if(countX == NpulsesRevolutionX){
          distanceX = comebackVarX/NpulsesRevolutionX;
          countX = 0;
          lcd.setCursor(0,1);
          lcd.print("X value: " + String(distanceX)  + " mm");
        }
      }
      mylcd.Print_String("X-position: " + String(feedXaxis) + " mm", 0, 162);
      delay(1000);
    }
  }
  return comebackVarZ;
}
```

Once the last repetition is complete, the tool comes back to the positioning pose and it is ready to execute the next work phase, that is the finishing.

3.3.2 Working phase: finishing stage

As said before, the aim of the finishing stage is to level the surface to obtain a better result.

The settings that must be changed with respect to the previous stage are only the feeds, both the X and Z axes. The latter quantity has been assumed constant, equal to 0,4 mm/rotation, and the rotational speed of the screw is calculated accordingly (code 45).

Code 45: Finishing stage

```
float feedXfinish = 0.010; //feed X for the finishing process
float feedZfinish = 0;

int completeProcess(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical complete cycle", 0, 0);

  feedZfinish = 0.4;
```

```
  rotationalSpeedScrew = (int) (feedZfinish*spindleSpeed*pitch);
  delay(1000);

  comebackVarZ = finishProcess(workingLength, rotationalSpeedScrew, feedXrough);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

Before performing the roughing process, the feed X has been reduced by a fixed amount, equal to 10 µm: this quantity is now added back to the previous feed X to obtain the overall amount of material to remove and to obtain the precise dimensions of the product.

The sequence of motion is the same as the roughing stage, but with the updated values regarding the feeds and the speeds involved (code 46).

Code 46: Finishing stage (detail)

```
long finishProcess(float length, int screwSpeed, float feedXaxis){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working finishing cycle:", 0, 0);
  delay(500);

  mylcd.Print_String("Position along the X-axis of the feed:", 0, 18);
  pulsesX = (feedXaxis + feedXfinish)*gearRatio*360/(pitch*angleX);
  internVarX = pulsesX;
  delayMotor = 2;
  for(int i=0; i<internVarX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    Forward();
    comebackVarX++;
    countX++;
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }
  mylcd.Print_String("X-position: "+String(feedXaxis + feedXfinish)+ " mm", 0, 36);
  delay(1000);

  pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;

  distanceCovered = 0.00;
  internVarZ = 0;

  halfperiodSpeed = 60*1e6;
  halfperiodSpeed = halfperiodSpeed/(2*NpulsesRevolutionZ);
  halfperiodSpeed = halfperiodSpeed/screwSpeed;
  periodSpeed = (int) halfperiodSpeed;

  digitalWrite(dirPin,HIGH);
  for(int i=0; i<pulses; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
```

```
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(periodSpeed);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(periodSpeed);
    comebackVarZ++;
    internVarZ++;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceCovered++;
      countZ = 0;
      lcd.home();
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  distanceCovered = length + extra_len;
  distanceZ = positioning + distanceCovered;
  mylcd.Print_String("Z-position: " + String(distanceCovered) + "/" +
String(length) + " mm", 0, 54);
  lcd.home();
  lcd.print("Z value: " + String(distanceZ) + " mm");
  delay(500);

  mylcd.Print_String("Return phase", 0, 90);
  mylcd.Print_String("Move back along X-axis", 0, 108);
  delayMotor = 2;
  for(int i=0; i<internVarX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    Backward();
    comebackVarX--;
    countX++;
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }
  delay(500);

  mylcd.Print_String("Move back along Z-axis", 0, 126);
  comebackVarZ = comebackVarZ - internVarZ;
  digitalWrite(dirPin,LOW);
  for(int i=0; i<internVarZ; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(halfT_screw);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(halfT_screw);
    comebackVarZ--;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
```

```
        distanceCovered++;
        distanceZ = comebackVarZ/NpulsesRevolutionZ;
        countZ = 0;
        lcd.home();
        lcd.print("Z value: " + String(distanceZ) + " mm");
      }
    }
    return comebackVarZ;
  }
```

Now the real finish process can be executed, following the same movements logic, already assumed for the roughing stage. Once the operation has been completed successfully, the retraction of the tool occurs, and all the final steps are executed to stop the work cycle safely.

3.4 Cylindrical turning: multidiameter process

The second process available on the processes list is the cylindrical turning, aiming to obtain a multidiameter product as depicted in the Fig. 61, again subdividing the whole process into a roughing and a finishing phase.

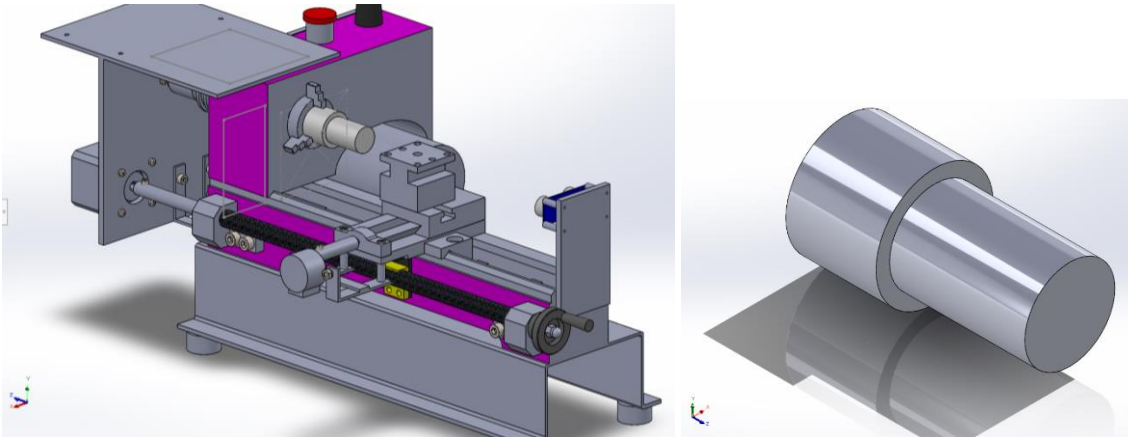


Figure 61: 3D model of the cylindrical multi diameters turning product

This turning option can be selected by the user pressing the 'B' code on the HMI keyboard and suggested on the monitor (Fig. 62).

Figure 62: Key to select the multidiameter cylindrical turning

Code 47: Multidiameter cylindrical turning

```
void loop(){
  startState = digitalRead(startPin); //acquisition of the signal from the start
connection
  secCheckState = digitalRead(secCheckPin);

  if(startState == HIGH && secCheckState == LOW){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Choose a working process ", 0, 0);
    mylcd.Print_String("from the following list:", 0, 18);
    mylcd.Print_String("> A: complete cylindrical turning with", 0, 36);
    mylcd.Print_String("roughing + finishing stages", 0, 54);
    mylcd.Print_String("> B: cylindrical turning for a", 0, 72);
    mylcd.Print_String("multidiameter component with", 0, 90);
    mylcd.Print_String("roughing + finishing stages", 0, 108);
    mylcd.Print_String("> C: conical turning with", 0, 126);
    mylcd.Print_String("roughing + finishing stages", 0, 144);
    mylcd.Print_String("> D: cylindrical roughing turning", 0, 162);
    mylcd.Print_String("Or * to exit", 0, 180);

    do{
      ok = 0;
      delay(50);
      startState = digitalRead(startPin);
      varMode = customKeypad.getKey();

      if(varMode && startState == HIGH){
        switch(varMode){
          case 'B': lcd.clear();
            ok = multidiameter();
            delay(1000);
            mylcd.Fill_Screen(BLUE);
            lcd.clear();
          break;
        }
      }else if(startState == LOW){
        Emergency();
        ok = 2;
      }
    }while(ok == 0);
  }else if(startState == LOW || secCheckState == HIGH){
    lcd.clear();
    mylcd.Fill_Screen(BLUE);
  }
}
```

The sequence of the movements of the tool is almost the same as the simple cylindrical turning, but specific evaluations must be considered in order to place correctly the tool to remove the right amount of material.

Considering the way the piece is mounted on the spindle and for a sake of simplicity, a constraint must be applied during the setup procedure. The user, counselled by the HMI instructions, must insert the smaller value of the final diameter firstly, and then the greater one. The former will be the shape on the right side of the product, while the latter the left side, and, at code level, they are labelled respectively with the 1 and 2 (code 48).

A suitable check is performed as soon as two values have been selected: in case of error (first diameter greater than the second one), the values must be re-inserted.

Code 48: Final diameters evaluation

```
float finalDiameter1 = 0;
```

```
float finalDiameter2 = 0;

int multidiameter(){
  do{
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
    mylcd.Print_String("Insert the  final diameter of the smaller", 0, 18);
    mylcd.Print_String("part of the piece with the keypad", 0, 36);
    mylcd.Print_String("between 0 mm and 30 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    finalDiameter1 = acquisitionSmallerDiameter(startDiameter);
    if(ok == 2){
      return;
    }
    delay(1000);

    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
    mylcd.Print_String("Insert the  final diameter of the greater", 0, 18);
    mylcd.Print_String("part of the piece with the keypad", 0, 36);
    mylcd.Print_String("between 0 mm and 30 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    finalDiameter2 = acquisitionGreaterDiameter(startDiameter);
    if(ok == 2){
      return;
    }
    delay(1000);
  }while(finalDiameter1 >= finalDiameter2);
}
```

Once the previous setting goes through correctly, the relative feed X values are evaluated (code 49).

At first stage the same function is executed, producing as output the two number of cycles required respectively for the smaller and for the greater part.

Code 49: Feed X values functions

```
int multidiameter(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
  mylcd.Print_String("Initial diameter: " + String(startDiameter) + " mm", 0, 18);
  mylcd.Print_String("Smaller initial diameter: " + String(finalDiameter1) + " mm",
0, 36);
  feedX1 = checkFeedXC(startDiameter, finalDiameter1);
  nCycles1 = numberCycles;

  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
  mylcd.Print_String("Initial diameter: " + String(startDiameter) + " mm", 0, 18);
  mylcd.Print_String("Greater initial diameter: " + String(finalDiameter2) + " mm",
0, 36);
  feedX2 = checkFeedXC(startDiameter, finalDiameter2);
  nCycles2 = numberCycles;
}

float checkFeedXC(float initialDiameter, float workDiameter){
  feedXint = (initialDiameter - workDiameter)/2;
  if(feedXint <= maxFeedX){
    numberCycles = 1;
    mylcd.Print_String(String(numberCycles) + " cycle needed", 0, 72);
```

```
    delay(2000);
  }else{
    numberCycles = (int)(feedXint/maxFeedX);
    ver = feedXint - numberCycles*maxFeedX;
    if(ver != 0){
      numberCycles++;
    }
    mylcd.Print_String(String(numberCycles) + " cycles needed", 0, 72);
    delay(2000);
  }
  return feedXint;
}
```

As further analysis, the real feed X values for the roughing processes are calculated, according to the following formulas. Regarding the greater section, the final roughing feed X is the original decreased by the fixed amount allocated to the finishing stage. Since the work on the greater part is executed before, the feed X for the smaller section must consider also the amount of material already taken away, according to:

$$FeedX\ greater\ section = \ feed\ X2 = feed\ X2_{original} - feed\ X_{finishing}$$

$$FeedX\ smaller\ section = feedX1 = feedX1_{original} - feedX2_{modified} - 2 * feedX_{finish}$$

Code 50: Feed X computation

```
feedX2 = feedX2 - feedXfinish;
feedX1 = feedX1 - feedX2 - 2*feedXfinish;
```

The same approach has been applied for the acquisition of the work lengths: now 3 parameters must be inserted by the user, since the whole piece is divided into 2 sections. Firstly, the overall work length is selected, therefore the 2 sub-lengths, respectively of the smaller and greater sections. Afterwards the check of the correctness is executed: if the sum of the previous two values does not coincide with the whole dimension, a new acquisition is demanded.

Code 51: Work lengths evaluation

```
int multidiameter(){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
  mylcd.Print_String("Insert the working length of the piece", 0, 18);
  mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 36);
  mylcd.Print_String("Press '#' to confirm the value", 0, 54);
  mylcd.Print_String("Press 'C' to delete the value and", 0, 72);
  mylcd.Print_String("re-insert a new one", 0, 90);
  workingLengthTotal = acquisitionWorkingLengthC();
  if(ok == 2){
    return;
  }
  delay(1000);

  do{
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
    mylcd.Print_String("Insert the working length of the smaller", 0, 18);
    mylcd.Print_String("part of the piece with the keypad", 0, 36);
    mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    workingLength1 = acquisitionWorkingLengthMultiSmaller();
    if(ok == 2){
```

```
      return;
    }
    delay(1000);

    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Cylindrical multidiameter cycle", 0, 0);
    mylcd.Print_String("Insert the working length of the greater", 0, 18);
    mylcd.Print_String("part of the piece with the keypad", 0, 36);
    mylcd.Print_String("with the keypad 0 mm and 200 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    workingLength2 = acquisitionWorkingLengthMultiGreater();
    if(ok == 2){
      return;
    }
    delay(1000);
  }while((workingLength1 + workingLength2) != workingLengthTotal);
}
```

3.4.1 Working phase: roughing phase

After the starting of the rotation of the spindle and the workpiece, the positioning of the tool takes place, where the feed X value considered corresponds to the one of the greater section. During its working phase, the first part on the right is machined as well: the result of this operation is a cylindrical piece with the dimension of the greater diameter.

The working phase of this part follows the exact same procedure as the one explained for the mono-dimensional cylindrical turning.

After this intermediate stage, the actual operation to obtain the smaller section can start. This operation obviously is performed along only the work length of the lower diameter part.

Firstly, compute the thickness between the greater and the smaller diameters, in order to establish the number of cycles needed. Afterwards, a new computation is executed before the real manufacturing, regarding the total feed from the initial diameter to the final diameter of this section.

Finally, one of the 2 functions is called, depending on the number of passes. If just one, the tool is moved according to the feed value and the work is machined (code 52).

Code 52: Roughing stage functions

```
int multidiameter(){
  //roughing cycle to obtain the smallest diameter
  feedX1 = feedX1 - feedX2 - 2*feedXfinish;
  if(feedX1 <= maxFeedX){
    nCycles1 = 1;
    feedX1 = feedX1 + feedX2 + feedXfinish;
    comebackVarZ = workingProcess1CycleSmall(workingLength1, rotationalSpeedScrew);
  }else{
    nCycles1 = (int)(feedX1/maxFeedX);
    ver = feedX1 - nCycles1*maxFeedX;
    if(ver != 0){
      nCycles1++;
    }
    feedX1 = feedX1 + feedX2 + feedXfinish;
    comebackVarZ = workingProcessMoreCyclesSmall(workingLength1,
rotationalSpeedScrew, feedX1);
  }
  if(ok == 2){
```

```
    return;
  }
  delay(1000);
}
```

On the other case, the loop structure is activated: the removal of the material occurs with a progressive maximum feed value for the intermediate passes, to complete the last repetition with the coverage of the residual depth.

Code 53: Multicycle roughing work

```
long workingProcessMoreCyclesSmall(float length, int screwSpeed, float feedXaxis){
  pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;

  for(int Ncycle=0; Ncycle<numberCycles; Ncycle++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(Ncycle+1), 0, 18);

    //re-positioning along the X-axis
    nC = Ncycle+1;
    if(nC < numberCycles){
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
      internVarX = (maxFeedX*nC + feedX2 + feedXfinish)*NpulsesRevolutionX/pitch;
      delayMotor = 2;
      for(int i=0; i<internVarX; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
        }
        Forward();
        comebackVarX++;
        countX++;
        if(countX == NpulsesRevolutionX){
          distanceX = comebackVarX/NpulsesRevolutionX;
          countX = 0;
          lcd.setCursor(0,1);
          lcd.print("X value: " + String(distanceX)  + " mm");
        }
      }
      mylcd.Print_String("X-position: " + String(maxFeedX) + " mm", 0, 54);
      delay(500);
    }else if(nC == numberCycles){
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
      pulsesX = (feedXaxis)*NpulsesRevolutionX/pitch;
      internVarX = pulsesX;
      delayMotor = 2;
      for(int i=0; i<internVarX; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
        }
        Forward();
        comebackVarX++;
        countX++;
        if(countX == NpulsesRevolutionX){
          distanceX = comebackVarX/NpulsesRevolutionX;
          countX = 0;
          lcd.setCursor(0,1);
```

```
          lcd.print("X value: " + String(distanceX)  + " mm");
        }
      }
      mylcd.Print_String("X-position: " + String(maxFeedX) + " mm", 0, 54);
      delay(500);
    }

    mylcd.Print_String("Z-position: " + String(distanceCovered) + " mm", 0, 72);
    mylcd.Print_String("Return phase", 0, 108);
    mylcd.Print_String("Move back along X-axis", 0, 126);
    mylcd.Print_String("Move back along Z-axis", 0, 144);
  }
  return comebackVarZ;
}
```

3.4.2 Working phase: finishing stage

Once the roughing stage terminates, both for the smaller and greater parts, the finishing phase starts. It is subdivided into 2 steps too.

As for the other processes, a fixed feed rate along the Z-axis has been engaged, equal to 0,4 mm/rotation.

At code level the function executed is the same: the difference between the two lies in the parameters of the work lengths and the movement along the transversal direction.

Firstly, the finishing process of the right part is manufactured (code 54): the work length inserted is just a fraction of the overall length, while the feed X value considers the depth of the smaller section (from the initial to the smaller final diameter).

Code 54: Multidiameter finishing phase (smallest section)

```
int multidiameter(){
  feedZfinish = 0.4;
  rotationalSpeedScrew = (int) (feedZfinish*spindleSpeed*pitch);

  //finishing process for the smallest diameter
  comebackVarZ = finishProcess(workingLength1, rotationalSpeedScrew, (feedX2 +
feedXfinish + feedX1));
  if(ok == 2){
    return;
  }
  delay(1000);
}

long finishProcess(float length, int screwSpeed, float feedXaxis){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working finishing cycle:", 0, 0);
  delay(500);

  mylcd.Print_String("Position along the X-axis of the feed:", 0, 18);
  pulsesX = (feedXaxis + feedXfinish)*gearRatio*360/(pitch*angleX);
  internVarX = pulsesX;
  delayMotor = 2;
  for(int i=0; i<internVarX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    Forward();
```

```
      comebackVarX++;
      countX++;
      if(countX == NpulsesRevolutionX){
        distanceX = comebackVarX/NpulsesRevolutionX;
        countX = 0;
        lcd.setCursor(0,1);
        lcd.print("X value: " + String(distanceX)  + " mm");
      }
    }
    mylcd.Print_String("X-position: "+String(feedXaxis + feedXfinish)+ " mm", 0, 36);
    delay(1000);

    pulses = NpulsesRevolutionZ*(length + extra_len)/pitch;

    distanceCovered = 0.00;
    internVarZ = 0;
    halfperiodSpeed = 60*1e6;
    halfperiodSpeed = halfperiodSpeed/(2*NpulsesRevolutionZ);
    halfperiodSpeed = halfperiodSpeed/screwSpeed;
    periodSpeed = (int) halfperiodSpeed;

    digitalWrite(dirPin,HIGH);
    for(int i=0; i<pulses; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      digitalWrite(stepPin,HIGH);
      delayMicroseconds(periodSpeed);
      digitalWrite(stepPin,LOW);
      delayMicroseconds(periodSpeed);
      comebackVarZ++;
      internVarZ++;
      countZ++;
      if(countZ == NpulsesRevolutionZ){
        distanceCovered++;
        countZ = 0;
        lcd.home();
        lcd.print("Z value: " + String(distanceZ) + " mm");
      }
    }
    distanceCovered = length + extra_len;
    distanceZ = positioning + distanceCovered;
    mylcd.Print_String("Z-position: " + String(distanceCovered) + "/" +
  String(length) + " mm", 0, 54);
    lcd.home();
    lcd.print("Z value: " + String(distanceZ) + " mm");
    delay(500);

    mylcd.Print_String("Return phase", 0, 90);
    mylcd.Print_String("Move back along X-axis", 0, 108);
    Serial.println(internVarX);
    delayMotor = 2;
    for(int i=0; i<internVarX; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      Backward();
      comebackVarX--;
      countX++;
```

```
    if(countX == NpulsesRevolutionX){
      distanceX = comebackVarX/NpulsesRevolutionX;
      countX = 0;
      lcd.setCursor(0,1);
      lcd.print("X value: " + String(distanceX)  + " mm");
    }
  }
  delay(500);

  mylcd.Print_String("Move back along Z-axis", 0, 126);
  comebackVarZ = comebackVarZ - internVarZ;
  Serial.println(comebackVarZ);
  digitalWrite(dirPin,LOW);
  for(int i=0; i<internVarZ; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
      return;
    }
    digitalWrite(stepPin,HIGH);
    delayMicroseconds(halfT_screw);
    digitalWrite(stepPin,LOW);
    delayMicroseconds(halfT_screw);
    comebackVarZ--;
    countZ++;
    if(countZ == NpulsesRevolutionZ){
      distanceCovered++;
      distanceZ = comebackVarZ/NpulsesRevolutionZ;
      countZ = 0;
      lcd.home();
      lcd.print("Z value: " + String(distanceZ) + " mm");
    }
  }
  return comebackVarZ;
}
```

Therefore, along the whole piece work length the tool moves to execute the finishing process also for the bigger section, where the position of the X-axis includes only its feed value (code 55).

Code 55: Multidiameter finishing phase (greatest section)

```
int multidiameter(){
  //finishing part
  feedZfinish = 0.4;
  rotationalSpeedScrew = (int) (feedZfinish*spindleSpeed*pitch);

  //finishing process for the greatest diameter and consider the length 1
  comebackVarZ = finishProcess(workingLengthTotal, rotationalSpeedScrew, feedX2);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

With this last operation, the work cycle for this kind of product can conclude successfully with the retraction of the carriage system.

3.5 Conical turning

The last mechanical process available on the lathe machine is the conical turning: the output product is a cone starting from a cylindrical raw part and with the conicity value specified by the user (Fig. 63). This process shows a more elaborated structure, with respect to the shape to work and the programming and control of the tool movement.

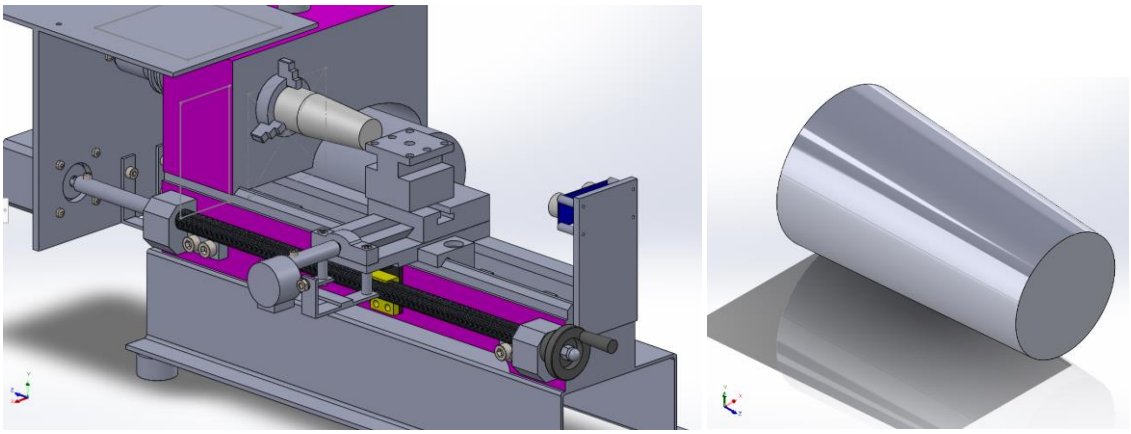


Figure 63: 3D model of the conical turning product

This turning option can be selected by the user pressing the 'C' code on the HMI keyboard and suggested on the HMI display (Fig. 64).

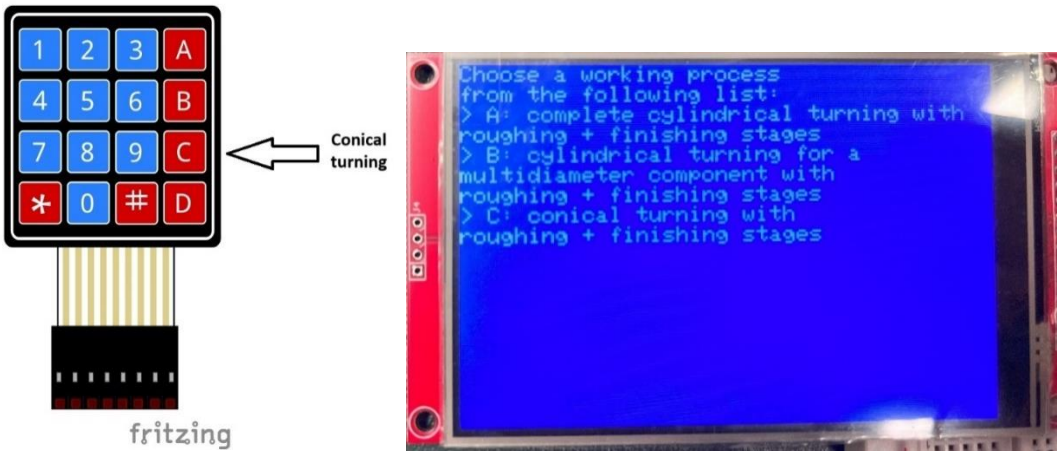


Figure 64: Key to select the conical turning

Code 56: Conical turning selection

```
void loop(){
  startState = digitalRead(startPin); //acquisition of the signal from the start
connection
  secCheckState = digitalRead(secCheckPin);

  if(startState == HIGH && secCheckState == LOW){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Choose a working process ", 0, 0);
    mylcd.Print_String("from the following list:", 0, 18);
    mylcd.Print_String("> A: complete cylindrical turning with", 0, 36);
    mylcd.Print_String("roughing + finishing stages", 0, 54);
    mylcd.Print_String("> B: cylindrical turning for a", 0, 72);
    mylcd.Print_String("multidiameter component with", 0, 90);
    mylcd.Print_String("roughing + finishing stages", 0, 108);
    mylcd.Print_String("> C: conical turning with", 0, 126);
```

```
    mylcd.Print_String("roughing + finishing stages", 0, 144);
    mylcd.Print_String("> D: cylindrical roughing turning", 0, 162);
    mylcd.Print_String("Or * to exit", 0, 180);

    do{
      ok = 0;
      delay(50);
      startState = digitalRead(startPin);
      varMode = customKeypad.getKey();

      if(varMode && startState == HIGH){
        switch(varMode){
          case 'C': lcd.clear();
            ok = conical();
            delay(1000);
            mylcd.Fill_Screen(BLUE);
            lcd.clear();
          break;
        }
      }else if(startState == LOW){
        Emergency();
        ok = 2;
      }
    }while(ok == 0);
  }else if(startState == LOW || secCheckState == HIGH){
    lcd.clear();
    mylcd.Fill_Screen(BLUE);
  }
}
```

The setting of the parameters follows the characteristic dimensions of the profile that will be machined, and this will be translated into a joint movement of the tool along both the motion axes. Indeed, the sequence of setup starts with the specification of the diameters, firstly of the smaller one and then with the greater, to obtain a conicity that enlarges moving from right to left of the lathe. For the sake of simplicity, only this conicity option has been considered.

With the specification of the cone length along the longitudinal direction, the stepper motors will be commanded in an alternating manner by sending a precise sequence of steps.

At code level, once the user has chosen to perform a conical turning, all the parameters required by the system are asked. The first information is obviously the main sizes of the product: the initial diameter and the two diameters at the two edges, called respectively “final begin diameter” on the right side with respect to the direction of movement of the tool, and “final end diameter” on the left side.

According to the requirement on the conicity, the code contains a size control, where the “final begin diameter” value must be strictly lower than the “final end diameter”. If the check returns a negative output, an error message is visualized, and a new selection of the final diameters is required.

Code 57: Conical final diameters evaluation

```
float finalDiameterBegin = 0;
float finalDiameterEnd = 0;

int conical(){
  do{
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Conical cycle", 0, 0);
    mylcd.Print_String("Insert the final diameter of the smaller", 0, 18);
```

```
    mylcd.Print_String("part of the cone with the keypad", 0, 36);
    mylcd.Print_String("between 0 mm and 30 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    finalDiameterBegin = acquisitionFinalDiameterConicalBegin(startDiameter);
    if(ok == 2){
      return;
    }
    delay(1000);

    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Conical cycle", 0, 0);
    mylcd.Print_String("Insert the final diameter of the greater", 0, 18);
    mylcd.Print_String("part of the cone with the keypad", 0, 36);
    mylcd.Print_String("between 0 mm and 30 mm", 0, 54);
    mylcd.Print_String("Press '#' to confirm the value", 0, 72);
    mylcd.Print_String("Press 'C' to delete the value and", 0, 90);
    mylcd.Print_String("re-insert a new one", 0, 108);
    finalDiameterEnd = acquisitionFinalDiameterConicalEnd(startDiameter);
    if(ok == 2){
      return;
    }
    delay(1000);

    if(finalDiameterBegin >= finalDiameterEnd){
      mylcd.Fill_Screen(BLUE);
      mylcd.Print_String("Conical turning cycle", 0, 0);
      mylcd.Print_String("ERROR!", 0, 18);
      mylcd.Print_String("Insert new values", 0, 36);
      mylcd.Print_String("for the final diameters", 0, 54);
    }
  }while(finalDiameterBegin >= finalDiameterEnd);
}
```

Later, as the other types of processes, the cutting speed, the theoretical rotational speed of the spindle, the feed along the Z-axis for the initial roughing part and the length of the piece must be inserted by the operator.

3.5.1 Working phase: preliminary cylindrical phase

Practically speaking, the work approach adopted consists into the subdivision of the whole work cycle into 2 main stages: a preliminary cylindrical and a subsequent conical turning. The former operation aims to manufacture a cylindrical work piece with the dimension of the greater final diameter, just to reduce the raw volume in a simpler and faster way. While the latter completes the process giving the effective conical contour.

This process subdivision is settled inside a specific function (code 58): its output is the number of cycles for each phase, by computing the feed along the X-axis corresponding to the two edges. Each feed value is converted into a number of passes, including the maximum feed X value and the residual amount.

Code 58: Feed X values computation

```
int checkConicity = 0;
float feedXbegin = 0;
int NmaxBegin = 0;
float residualFeedXbegin = 0;

float feedXend = 0;
int NmaxEnd = 0;
```

```
float residualFeedXend = 0;

bool conicity = 0;
float feedXcyclindrical = 0;
int numberCyclesCyl = 0;

float deltaX = 0;
float deltaZ = 0;
float feedXconicalBegin = 0;
float feedXconicalEnd = 0;
int NcyclesConicalMax = 0;

int conical(){
  checkFeedXconical();
}

void checkFeedXconical(){
  checkConicity = 0;
  do{
    feedXbegin = (startDiameter - finalDiameterBegin)/2;
    feedXbegin = feedXbegin - feedXfinish;
    NmaxBegin = (int) feedXbegin/maxFeedX;
    residualFeedXbegin = feedXbegin - NmaxBegin*maxFeedX;

    feedXend = (startDiameter - finalDiameterEnd)/2;
    feedXend = feedXend - feedXfinish;
    NmaxEnd = (int) feedXend/maxFeedX;
    residualFeedXend = feedXend - NmaxEnd*maxFeedX;

    //cylindrical turning
    if(finalDiameterBegin < finalDiameterEnd){
      conicity = 1;
      feedXcyclindrical = NmaxEnd*maxFeedX;
      numberCyclesCyl = NmaxEnd;

      deltaX = (finalDiameterEnd/2) - (finalDiameterBegin/2);
      deltaZ = workingLength;
      feedXconicalBegin = feedXbegin - feedXcyclindrical;
      feedXconicalEnd = residualFeedXend;
      NcyclesConicalMax = NmaxBegin - NmaxEnd;
      checkConicity = 1;
    }
  }while(checkConicity == 0);
}
```

Subsequently, if the feed X of the end diameter is greater or equal than the maximum feed of one surface treatment pass, a preliminary cylindrical turning is required, otherwise this phase is skipped, and the conical phase can start immediately. The number of repetitions corresponds to the full passes, previously computed.

The development of the cylindrical turning is almost equivalent to the one performed for the standard cylindrical work cycles, about the movements of the tool of positioning and back motion.

Again, two different functions are used based on the number of repetitions required, one or more (code 59).

Code 59: Preliminary cylindrical turning

```
int conical(){
  if(numberCyclesCyl == 1){
    comebackVarZ = workingProcess1Cycle(workingLength,rotationalSpeedScrew);
  }else if(numberCyclesCyl > 1){
```

```
    comebackVarZ = workingProcessMoreCyclesCyl(workingLength,rotationalSpeedScrew);
  }
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

A difference with the standard multiple work cycle stands inside the last pass check: during this stage, although in the other processes the residual feed X is removed, for the preliminary cylindrical turning for conical products only maximum feeds are considered. It would be taken into consideration during the conical operation (code 60).

Code 60: Multicycle cylindrical work phase

```
long workingProcessMoreCyclesCyl(float length, int screwSpeed){
  for(int Ncycle=0; Ncycle<numberCyclesCyl; Ncycle++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(Ncycle+1), 0, 18);
    mylcd.Print_String("Z-position: " + String(distanceCovered) + " mm", 0, 36);
    mylcd.Print_String("Return phase", 0, 72);
    mylcd.Print_String("Move back along X-axis", 0, 90);
    mylcd.Print_String("Move back along Z-axis", 0, 108);

    //re-positioning along the X-axis
    if((Ncycle+1) < numberCyclesCyl){
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 144);
      internVarX = maxFeedX*NpulsesRevolutionX*(Ncycle+2);
      delayMotor = 2;
      for(int i=0; i<internVarX; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
        }
        Forward();
        comebackVarX++;
        countX++;
        if(countX == NpulsesRevolutionX){
          distanceX = comebackVarX/NpulsesRevolutionX;
          countX = 0;
          lcd.setCursor(0,1);
          lcd.print("X value: " + String(distanceX)  + " mm");
        }
      }
      mylcd.Print_String("X-position: " + String(maxFeedX) + " mm", 0, 162);
      delay(1000);
    }
  }
  return comebackVarZ;
}
```

As already done for the multidiameter turning, the angular speed of the lead screw system depends on the angular velocity of the spindle; this last parameter has been computed considering the average value between the begin and end final diameters. Final step is to calculate the half period of the square wave signal for the stepper motor.

3.5.2 Working phase: conical phase

The core of the process is the logic referred to the joint movement of the stepper motors, manufacturing the real conical profile.

Code 61: Conical work cycle

```
int conical(){
  comebackVarZ = conicalTurning(rotationalSpeedScrew);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

Some calculations must be done to handle the sequence of pulses to send to each component.

The main concept is to command the motion of the tool along paths, that are parallel to the theoretical profile of the cone. The feed values at the begin and end edges are acquired from the initial feeds and the cylindrical amount already removed, as shown in the code 62. The final line computes the number of passes where 0,5 mm of depth is applied.

Code 62: Conical turning features

```
deltaX = (finalDiameterEnd/2) - (finalDiameterBegin/2);
deltaZ = workingLength;
feedXconicalBegin = feedXbegin - feedXcyclindrical;
feedXconicalEnd = residualFeedXend;
NcyclesConicalMax = NmaxBegin - NmaxEnd;
```

In order to accomplish this approach, the displacements along both the motion axes are computed, as guide for the movement:

$$\Delta x = \frac{final\ end\ diameter - final\ begin\ diameter}{2}\ and\ \Delta z = working\ length$$

Since the stepper motors are digital machines commanded by a digital program, they cannot move along a continuous contour: the conical edge will be approximated with a profile made by a certain number of discrete steps. If the number and the amplitude of the steps are chosen correctly, the final profile can be like an ideal continuous conical profile. The roughness depends strongly on the resolution of the stepper motors as well: a smaller unit angle means a lower movement of the output shaft.

The whole procedure is divided into two sections (code 63): the first one includes a loop function to repeat a parallel joint motion as many times as the number of the maximum feed passes requires; while a further work cycle is executed to complete and remove all the residual material along the contour.

The for-loop structure starts by positioning the tool along the X direction in a progressive way, cumulating the feed of the begin final diameter to remove.

Code 63: Conical turning functions (1)

```
int pulsesExtra = 0;

long conicalTurning(int screwSpeed){
  halfperiodSpeed = 60*1e6;
```

```
  halfperiodSpeed = halfperiodSpeed/(2*NpulsesRevolutionZ);
  halfperiodSpeed = halfperiodSpeed/screwSpeed;
  periodSpeed = (int) halfperiodSpeed;

  for(int k=0; k<NcyclesConicalMax; k++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(k+1), 0, 18);

    //move the motor X according to the feed X
    mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
    internVarX = maxFeedX*NpulsesRevolutionX*(k + numberCyclesCyl + 1);
    delayMotor = 2;
    for(int i=0; i<internVarX; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      Forward();
      comebackVarX++;
    }
    mylcd.Print_String("X-position: " + String(maxFeedX*(k + numberCyclesCyl + 1))
+ " mm", 0, 54);
    delay(500);

    //move the motor Z along the extra-length
    pulsesExtra = NpulsesRevolutionZ*extra_len/pitch;
    internVarZ = 0;
    digitalWrite(dirPin,HIGH);
    for(int j=0; j<pulsesExtra; j++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      digitalWrite(stepPin,HIGH);
      delayMicroseconds(halfT_screw);
      digitalWrite(stepPin,LOW);
      delayMicroseconds(halfT_screw);
      comebackVarZ++;
      internVarZ++;
    }
    mylcd.Print_String("Z-position: " + String(extra_len) + " mm", 0, 72);
  }
}
```

Start by computing the number of pulses useful for the motors to cover the two above displacements:

$$Pulses\ motor\ Z = n°\ of\ pulses\ per\ revolution\ Z * \Delta z = 800 * \Delta z$$

$$Pulses\ motor\ X = n°\ of\ pulses\ per\ revolution\ X * \Delta x = 512 * \Delta x$$

The amplitude of each step for the motor Z is computed by the following formulas:

$$Partial\ pulses\ motor\ Z = \frac{pulses\ motor\ Z}{pulses\ motor\ X}$$

$$Residual\ pulses\ motor\ Z = pulses\ motor\ Z - pulses\ motor\ X * partial\ pulses\ motor\ Z$$

The logic consists of sending a sequence of pulses to the motor Z of amplitude equal to the quantity “partial pulses motor Z” followed by a single pulse of the motor X; since the tool must follow the conicity from a smaller to a larger diameter, the Z stepper motor advances and the X motor retreats (code 64).

With this procedure, the work length is subdivided into a certain number of steps of equal amplitude. A peculiarity is related to the first “residual pulses motor Z”: the amplitude of each step is increased by a unit pulse. Suitable counter variables are defined to trace this evolution and to stop the increase of the Z steps.

Code 64: Conical turning functions (2)

```
long pulsesZconical = 0;
long pulsesXconical = 0;
int partialPulses= 0;
int residualPulses = 0;

int incr = 0;
int incrRes = 0;
int n = 0;

long conicalTurning(int screwSpeed){
  pulsesZconical = NpulsesRevolutionZ*deltaZ;
  pulsesXconical = NpulsesRevolutionX*deltaX;

  if(pulsesZconical >= pulsesXconical){
    partialPulses = pulsesZconical/pulsesXconical;
    residualPulses = pulsesZconical - pulsesXconical*partialPulses;

    for(int k=0; k<NcyclesConicalMax; k++){
      mylcd.Fill_Screen(BLUE);
      mylcd.Print_String("Working roughing cycle:", 0, 0);
      mylcd.Print_String("Cycle number: " + String(k+1), 0, 18);
      mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
      mylcd.Print_String("X-position: " + String(maxFeedX*(k + numberCyclesCyl +
1)) + " mm", 0, 54);
      mylcd.Print_String("Z-position: " + String(extra_len) + " mm", 0, 72);

      delayMotor = 2;
      digitalWrite(dirPin,HIGH);
      incr = 0;
      incrRes = 0;
      do{
        if(incrRes < residualPulses){
          n = partialPulses + 1;
        }else{
          n = partialPulses;
        }
        for(int b=0; b<n; b++){
          startState = digitalRead(startPin);
          secCheckState = digitalRead(secCheckPin);
          if(startState == LOW || secCheckState == LOW){
            Emergency();
            ok = 2;
            return;
          }
          digitalWrite(stepPin,HIGH);
          delayMicroseconds(periodSpeed);
          digitalWrite(stepPin,LOW);
          delayMicroseconds(periodSpeed);
          comebackVarZ++;
          internVarZ++;
        }
```

```
        Backward();
        comebackVarX--;
        incr++;
        incrRes++;
      }while(incr < pulsesXconical);
      mylcd.Print_String("Z-position: " + String(deltaZ+extra_len) + " mm", 0, 90);
    }
  }
}
```

During the work motion, the value of the counter “internVarX” increases from the positioning pose to the X coordinate corresponding to the feed X value; while the counter “comebackVarX” records the global movement of the tool alongside the X axis, and it increases or decreases according to the direction of the X stepper motor. In the specific case of the conical turning, at a primary stage, it increases for the positioning, afterwards it reduces due to the movement of the tool according to the conicity.

As soon as a work cycle is complete, a check for the return phase is required (code 65). The tool will move forward or backward, by the comparison between the number of pulses referred to the movement along the X-axis of the cone and the pulses required for the positioning along the X-axis before the work cycle. The difference between the previous variables determines the number of steps to send to the motor.

Code 65: Conical turning functions (3)

```
long conicalTurning(int screwSpeed){
  for(int k=0; k<NcyclesConicalMax; k++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(k+1), 0, 18);
    mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
    mylcd.Print_String("X-position: " + String(maxFeedX*(k + numberCyclesCyl + 1))
+ " mm", 0, 54);
    mylcd.Print_String("Z-position: " + String(extra_len) + " mm", 0, 72);
    mylcd.Print_String("Z-position: " + String(deltaZ+extra_len) + " mm", 0, 90);

    //move back the tool
    mylcd.Print_String("Return phase", 0, 126);
    mylcd.Print_String("Move back along X-axis", 0, 144);
    if(internVarX <= pulsesXconical){
      delayMotor = 2;
      n = pulsesXconical - internVarX;
      for(int i=0; i<n; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
          ok = 2;
          return;
        }
        Forward();
        comebackVarX++;
      }
    }else if(internVarX > pulsesXconical){
      delayMotor = 2;
      n = internVarX - pulsesXconical;
      for(int i=0; i<n; i++){
        startState = digitalRead(startPin);
        secCheckState = digitalRead(secCheckPin);
        if(startState == LOW || secCheckState == LOW){
          Emergency();
```

```
          ok = 2;
          return;
        }
        Backward();
        comebackVarX--;
      }
    }
    delay(500);
  }
}
```

The return phase concludes with the re-positioning of the tool along the longitudinal axis (code 66).

Code 66: Conical turning functions (4)

```
long conicalTurning(int screwSpeed){
  for(int k=0; k<NcyclesConicalMax; k++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(k+1), 0, 18);
    mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
    mylcd.Print_String("X-position: " + String(maxFeedX*(k + numberCyclesCyl + 1))
+ " mm", 0, 54);
    mylcd.Print_String("Z-position: " + String(extra_len) + " mm", 0, 72);
    mylcd.Print_String("Z-position: " + String(deltaZ+extra_len) + " mm", 0, 90);
    mylcd.Print_String("Return phase", 0, 126);
    mylcd.Print_String("Move back along X-axis", 0, 144);

    mylcd.Print_String("Move back along Z-axis", 0, 162);
    digitalWrite(dirPin,LOW);
    for(int i=0; i<internVarZ; i++){
      startState = digitalRead(startPin);
      secCheckState = digitalRead(secCheckPin);
      if(startState == LOW || secCheckState == LOW){
        Emergency();
        ok = 2;
        return;
      }
      digitalWrite(stepPin,HIGH);
      delayMicroseconds(halfT_screw);
      digitalWrite(stepPin,LOW);
      delayMicroseconds(halfT_screw);
      comebackVarZ--;
    }
    delay(500);
  }
}
```

As said before, the work operation terminates with the execution of a conical motion to finish the removal of material and to obtain a final product stuck to the desired shape (code 67). The logical and practical sequence is the same as the one above explained. To machine this stage, a check on the value of the residual feed in correspondence of the begin edge is required: if it is null, this phase can be skipped.

Code 67: Conical turning functions (5)

```
long conicalTurning(int screwSpeed){
  for(int k=0; k<NcyclesConicalMax; k++){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(k+1), 0, 18);
    mylcd.Print_String("Position along the X-axis of the feed:", 0, 36);
```

```
    mylcd.Print_String("X-position: " + String(maxFeedX*(k + numberCyclesCyl + 1))
+ " mm", 0, 54);
    mylcd.Print_String("Z-position: " + String(deltaZ+extra_len) + " mm", 0, 90);
    mylcd.Print_String("Return phase", 0, 126);
    mylcd.Print_String("Move back along X-axis", 0, 144);
    mylcd.Print_String("Move back along Z-axis", 0, 162);
    ...
  }

  //last cycle where the feed X begin is not maximum
  if(residualFeedXbegin != 0){
    mylcd.Fill_Screen(BLUE);
    mylcd.Print_String("Working roughing cycle:", 0, 0);
    mylcd.Print_String("Cycle number: " + String(NcyclesConicalMax+1), 0, 18);
    ...
  }
  return comebackVarZ;
}
```

3.5.3 Working phase: finishing phase

To conclude the whole mechanical operation, a last finishing pass is machined to make uniform the surface of the cone. As for the other processes, a fixed limited feed X-axis value has been defined, also for the conical the assumed feed is equal to 10 µm: this quantity has been subtracted to the original feed X values corresponding to the begin and end final diameters. When the finishing function is called, the inverse operation is executed.

Also regarding to the feed Z-axis value engaged, the approach remains the same as the two previous mechanical processes.

Code 68: Finishing phase for the conical turning

```
int conical(){
  feedXfinish = 0.010; //[mm]
  feedZfinish = 0.4; //[mm/rotation]
  rotationalSpeedScrew = (int) (feedZfinish*spindleSpeed*pitch);
  delay(1000);

  comebackVarZ = finishProcessConical(rotationalSpeedScrew);
  if(ok == 2){
    return;
  }
  delay(1000);
}
```

Since the tool must follow the cone profile, inside this work function has been kept the same sequence of movements for the stepper motors of the roughing phase (code 69).

Code 69: Finishing phase for the conical turning (detail)

```
long finishProcessConical(int screwSpeed){
  mylcd.Fill_Screen(BLUE);
  mylcd.Print_String("Working finishing cycle:", 0, 0);
  mylcd.Print_String("Position along the X-axis of the feed:", 0, 18);

  internVarX = (feedXbegin + feedXfinish)*NpulsesRevolutionX/pitch;
  delayMotor = 2;
  for(int i=0; i<internVarX; i++){
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      ok = 2;
```

```
      return;
    }
    Forward();
    comebackVarX++;
  }
  mylcd.Print_String("X-position: "+String(feedXbegin + feedXfinish)+ " mm", 0,
36);
  delay(500);

  //move the motor Z along the extra-length
  pulsesExtra = NpulsesRevolutionZ*extra_len/pitch;
  mylcd.Print_String("Z-position: " + String(extra_len) + " mm", 0, 54);

  //move before the motor Z then perform the sequence of the motor X
  mylcd.Print_String("Z-position: " + String(deltaZ+extra_len) + " mm", 0, 72);

  //move back the tool
  mylcd.Print_String("Return phase", 0, 108);
  mylcd.Print_String("Move back along X-axis", 0, 126);
  if(internVarX < pulsesXconical){
    ...
  }else if(internVarX >= pulsesXconical){
    ...
  }

  mylcd.Print_String("Move back along Z-axis", 0, 144);
  return comebackVarZ;
}
```

4. Layout of the control of the DC motor

The second part of the project deals with the power supply and control of the electric motor, responsible for the rotation of the workpiece during the turning process.

This component belongs to the family of the direct current (DC) electrical motor [15]. The main physical principle is the interaction between an electrical direct current and a magnetic field, and this phenomenon represents the source of the motion of the motor shaft.

Inside this group of motors, a subdivision is related to the principle of generation of the magnetic field, and they refer to the real components inside the motor body generating it: permanent magnets or electromagnets DC motors. The former contains magnets which constantly generate a magnetic field in the stator; while the latter contains magnets which are able to produce a magnetic field when a current flows inside the wirings of the stator. Although the permanent magnets generate a constant field around them and not tuneable with an external intervention, this type of components does not require an additional electrical circuit: the only one mandatory is the network for the current and voltage control for the motor armature, which has a direct influence on the rotational speed and output torque.

The Fig. 65 illustrates a motor prototype of this kind:

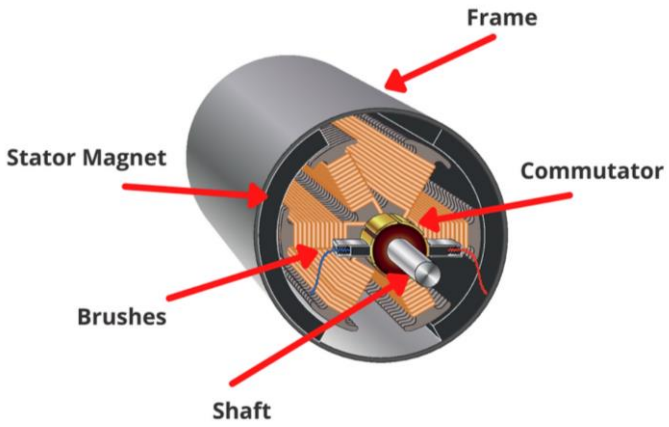


Figure 65: PMDC motor structure

Attached to the stator, there is the presence of two permanent magnets (underlined by the absence of the excitation wiring), and the rotor with the armature circuit and the output shaft. In the specific application of the lathe, this last element is connected to a system of pulleys and belt useful to transmit the mechanical power (Fig. 66).

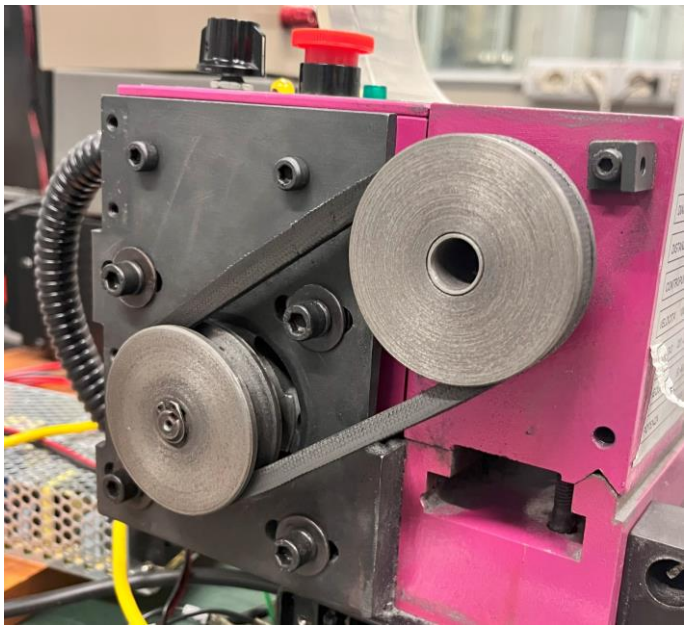


Figure 66: Mechanical system for the rotation transmission

Mathematically speaking, the behaviour of this DC motor can be described by these two equations, linking the main physical quantities of the system.

$$Armature\ voltage = V_a = R_a * I_a + L_a * \frac{dI_a}{dt} + E_a, where\ E_a = K_E * \omega$$

$$Torque = C_a = K_T * I_a$$

The armature voltage V_a is the voltage set to the brushes side of the motor, and this value can be tuned by an external circuit of interface between the microcontroller and the motor itself. By its control, the visible effect is a variable rotational velocity of the rotor and the shaft as well.

This electromechanical conversion, from the electrical power of the wiring to the mechanical output power, depends on the structure and internal features of the motor. Looking at the equations above, both the electrical and the mechanical characteristics are involved: for example, the current flowing across the armature depends on the external voltage applied and by the constant K, which contains information on the structure of the system.

Starting from the presence of the motor on the lathe machine, this section of the thesis is aimed to design, realize and control an interface network between the microcontroller (the Arduino board) and the DC motor itself.

Since on the main logic program a certain number of different mechanical processes has been developed, also with the need of handling various sizes of the products, the main goal of this circuit is to generate a variable voltage and so a variable speed. In order to develop this task, a preliminary simple network has been realized involving low power, just to test the correctness of the logic. Subsequently, an improvement of this network is designed.

The method applied for the control of the velocity is the Pulse Width Modulation (PWM) approach.

The origin approach was based on a direct and analogic control of the speed: through the command of a potentiometer, the user could set the desired value of the velocity for the spindle, but there was not a measurement instrument of the real value.

To take a step forward with respect to the original control method, a feedback control loop has been implemented in order to include the digital world of the microprocessors and to obtain an automatic behaviour of the machine.

From the Arduino code of the processes, the desired angular speed value is computed during the setup stage and sent to the other Arduino board: this number represents the reference for the loop scheme (Fig. 67). The feedback branch takes into account the acquisition of the real position of the spindle through an encoder: with some elaborations, this instrument gives as output the actual rotational velocity, which is compared with the target one. Subsequently, this error value is elaborated with a suitable proportional-integral-derivative (PID) controller to generate the input command for the PWM circuit and finally the voltage signal is sent to the motor.

This method can be pictured as follows, where the black elements are ones proper of the feedback control scheme and the red arrows point the features managed by the Arduino.

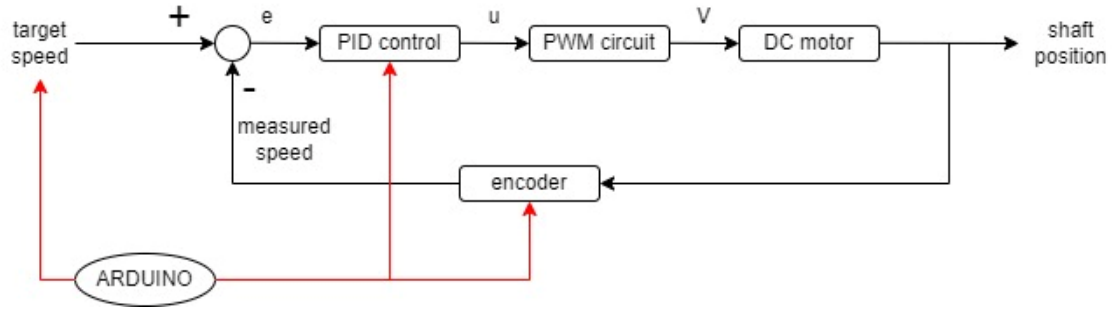


Figure 67: Feedback control loop scheme

This approach includes both the handling of analogic and digital signals, for example the voltage (analogic quantity) from the encoder must be converted into a digital information, manageable by the digital microcontroller.

4.1 Electrical circuit of the DC motor

The Pulse Width Modulation (PWM) technique is a particular control method to drive the electrical motors, and the goal is to tune the electrical power injected into the motor itself. In the case of the DC type, the controlled quantities are the armature voltage and the current.

This circuit includes as leading elements the Arduino board, as the source of the control signal, and the metal-oxide-semiconductor field-effect transistor (MOSFET), which links the microcontroller and the motor [16].

A MOSFET is an active electronic component, and its working principle is based on the physical properties of the semiconductor materials. These, very widespread in the electronic world, can be classified into two categories: n-type and p-type. The first group includes the materials where at atomic level there is a predominance of negative charges (electrons) over the positive ones (holes), while the opposite situation occurs for the p-type category.

The transistor is constructed with this semiconductor material and internally there are present both regions of n-type and p-type: when they are placed together, two electrical terminals arise, called “drain” and “source” (as depicted in Fig. 68). Between these connections, an electrical channel for the flowing current is created and an external input control signal modulates its thickness, which influences the value of the current. Both n-type and p-type channel devices exist: for this project only the n-type has been described and used.

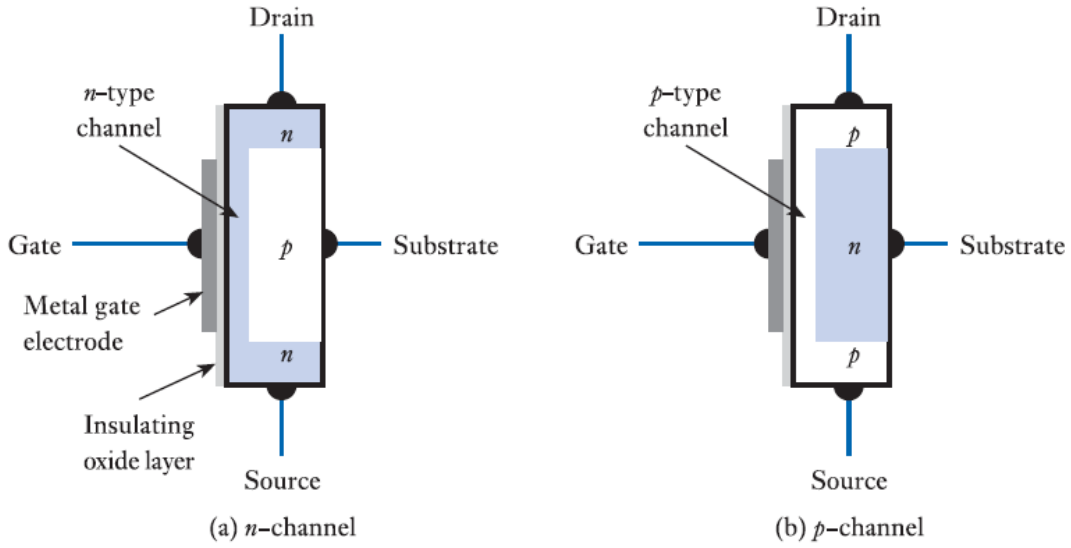


Figure 68: MOSFET distinction

The “metal-oxide” transistors are so called since the connection of the gate terminal is insulated from the conducting channel by a layer of insulating oxide.

The last connection regards the “substrate”: since the transistor belongs to the n-channel type, the presence of a p-type semiconductor piece is needed, so the two n-type regions of the “drain” and the “source” electrodes are created. The substrate is often internally connected to the source to obtain a 3 external connections device.

The channel between the drain and the source represents a conduction path between these two electrodes and therefore permits a flow of current. With zero volts applied to the gate, a voltage applied between the drain and the source will produce an electric field that will cause the mobile charge carriers in the channel to flow, thus creating a current. The magnitude of this current will be determined by the applied voltage and the number of charge carriers available in the channel.

The application of a voltage to the gate of the device will affect the number of charge carriers in the channel and hence the flow of electricity between the drain and source. The metal gate electrode and the semiconductor channel represent two conductors separated by an insulating layer. The construction therefore resembles a capacitor and the application of a voltage to the gate will induce the build-up of charge on each side. If the gate of an n-channel MOSFET is made positive with respect to the channel, this will attract electrons to the channel region, increasing the number of mobile charge carriers and increasing the apparent thickness of the channel. Under these circumstances the channel is said to be enhanced. If the gate is made negative, this will repel electrons from the channel, reducing its thickness. Here, the channel is said to be

depleted. Thus, the voltage on the gate directly controls the effective thickness of the channel and the resistance between the drain and the source.

The junction between the p-type substrate and the various n-type regions represents a p-n junction, which will have the normal properties of a semiconductor diode. However, in normal operation the voltages applied ensure that this junction is always reverse biased so that no current flows. For this reason, the substrate can be largely ignored when considering the operation of the device.

There exist MOSFETs which behave in the way just described, called depletion-enhancement MOSFETs, but also other devices, enhancement-MOSFETs, that have the peculiarity of enhancing the channel when a positive voltage is applied to the gate, and have no channel in absence of any gate potential and so any current can flow. Again, also in this case, the application of a positive voltage to the gate will attract electrons and repel holes from the region around the gate itself to form a conducting n-type channel. The last description reflects the device used inside the lathe circuit.

The transistor used inside the interface network is the “IRL530N” power enhancement MOSFET (in Fig. 69 the electrical symbol and the pinout).

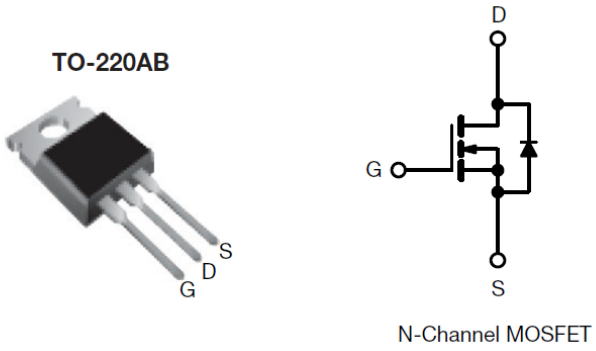


Figure 69: IRL530N MOSFET

Practically and as in the scheme, the 3 external connectors, in the order “gate”, “drain” and “source”, have been connected respectively to the control signal coming from the Arduino, the negative terminal of the DC motor and the last one to the ground, common to the Arduino ground.

The control signal is a square wave voltage, which is generated by the Arduino board, of amplitude 0-5 V and a variable duty cycle, which is the amount of time where the signal is up to 5 V. The frequency is fixed and set by the PWM channels of the microcontroller: for this application, the value is imposed to 31250 Hz, corresponding to a wave period of 32 µs (the duty cycle ranges between 0% and 100% of the period value).

The Fig. 70 schemes the connections of the transistor with the voltages generators and the schematic of the motor.

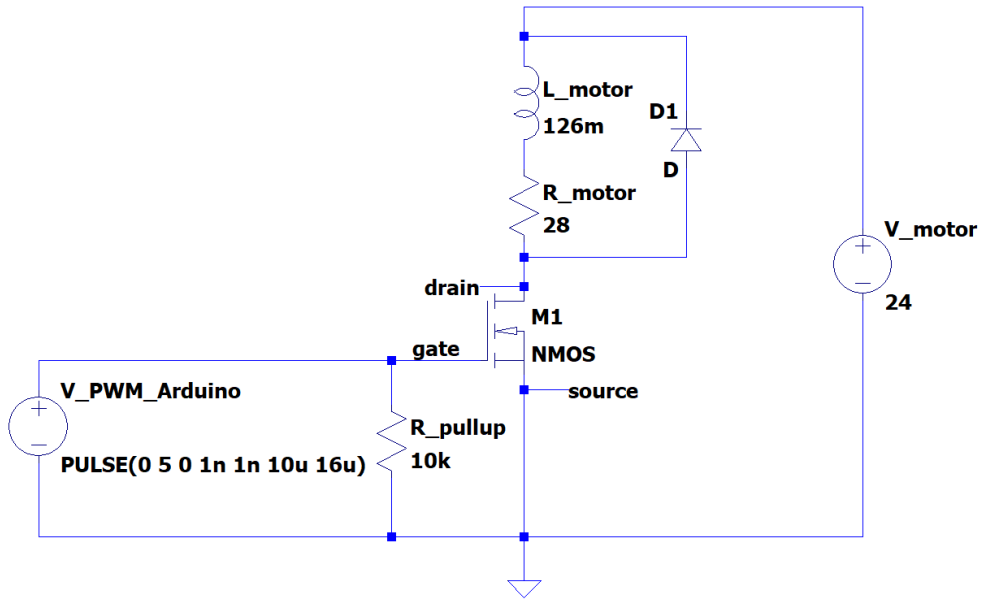


Figure 70: PWM circuit

By modulating the switching phases of the transistor, the output effect is a non-constant voltage applied in time domain to the motor, leading to a variable shaft speed, although the input voltage is constant to 24 V (from an external voltage supplier). For example, if the duty cycle of the gate voltage is 60%, this means that the current and the supply voltage are injected to the armature only the 60% of the total period, and so on the average the resultant voltage is almost the 60% of the maximum value.

As mentioned before, the command of the duty cycle is performed automatically by the Arduino on the basis of the theoretical value computed with the parameter formulas. Thanks to the application of the encoder and of the PID controller, an automated adjustment is possible to keep the velocity as steady as possible to the target and limited within a certain range. This tolerance has been necessary introduced, since the approximations of the measurements and physical limitations of the machines.

The just described electrical supply circuit is one of all the possible solutions. This has been chosen for its simplicity in terms of components used, but one drawback is the limited range of spindle angular speeds achievable, due to the constant low input armature voltage.

Other approaches for the driving circuit can be various, for example a boost converter or a buck converter. The first one, based on the energy conservation across the input and output stages, is able to convert a low input voltage into a higher output value, lowering the current flowing through. On the contrary, the buck converter encloses principles from both the previous solutions: it is always based on the input-output power conservation, but at the input stage a greater voltage is applied to modulate it and produce a lower output voltage, depending on the duty cycle level of the control signal.

Later a theoretical and simulative design has been developed, as feasible alternative of the initial solution.

4.2 Speed regulation of the DC motor with the Arduino board

The following section is dedicated to the description of the Arduino code: it has been developed for the control of the rotational speed of the direct current motor, which is connected to the spindle, holding the working piece. The Arduino board must handle at the same time the communication with the main Arduino microcontroller, where the programme of the working processes is uploaded, and the right signal to send to the electrical PWM circuit interfacing the digital control and the motor. The logic is subdivided into subfunctions fulfilling the multiple tasks.

Before getting into the detailed description of the Arduino functions related to the speed management, a sequence of procedures must be followed to prepare suitably the machine to operate. These commands are related to the reading and writing of some signals useful to set up the machine, regarding the human-machine interface and the security of the user, helped by a series of indicator lights.

4.2.1 Preliminary operations

The first function met inside the program deals with the interface between the user and the machine, defining the mode of the preliminary settings. Once the software is uploaded on the microcontroller, the loop function starts executing and it waits for the switching on of the lathe, signal coming from the suitable command element (code 70).

An additional control regards the position of the security window over the spindle grips and the metal piece under work: this must be open because the preparation of the mechanical operation must follow a precise sequence, also including the check of the window, whether open or close. Going through these steps, in the case it is already closed, the rotation of the chuck can start due to a misbehaviour or misreading of the sensor causing errors and in the worst-case injuries for the operator.

Code 70: Start circuit and security window check

```
//Start with a led
#define startPin 22 //pin for the start signal
int startState = 0; //variable for the state of the start signal
#define startLed 24 //pin for the led of the start

//check if the spindle window is close or open
#define secCheckPin 26 //pin for the security window over the spindle
int secCheckState = 0; //variable for the state of the security window over the
spindle
#define pauseLed 28 //pin for the led of the spindle window

#define stopLed 32
#define emergencyLed 44

#define pwmPin 6 //pin for the PWM signal (frequency = 488 Hz and the default
frequency is 31250 Hz)

void setup(){
  //enabling of the pins for the reading of the start and PWM signals
  pinMode(startPin, INPUT_PULLUP);
  pinMode(startLed, OUTPUT);
  pinMode(secCheckPin, INPUT_PULLUP);
  pinMode(pauseLed, OUTPUT);
  pinMode(stopLed, OUTPUT);
  pinMode(emergencyLed, OUTPUT);
```

```
  pinMode(pwmPin, OUTPUT);

  lcdSpeed.begin(16,2);
  lcdSpeed.clear();
}

void loop(){
  lcdSpeed.clear();
  digitalWrite(startLed,LOW); //keep the green-start led off
  digitalWrite(pauseLed,LOW); //keep the blue-pause led off
  digitalWrite(stopLed,LOW); //turn on the yellow-stop light
  digitalWrite(emergencyLed,LOW); //keep the red-emergency led off

  startState = digitalRead(startPin); //acquisition of the signal from the start
  secCheckState = digitalRead(secCheckPin); //acquisition of the signal from the
security window over the spindle

  if(startState == HIGH && secCheckState == LOW){ //switch on the machine and the
window must be open
    setMotor(0,pwmPin);
    digitalWrite(startLed,LOW);
    digitalWrite(pauseLed,LOW);
    digitalWrite(stopLed,HIGH); //turn on the yellow-stop light

    lcdSpeed.clear();
    lcdSpeed.print("Machine ready");
    lcdSpeed.setCursor(0,1);
    lcdSpeed.print("for a process");
  }
}
```

The next step regards the enabling of the process and the acquisition of the target value for the rotation of the system DC motor-spindle.

The first signal is exchanged by the two Arduino boards, and it is of digital/Boolean type: it turns “HIGH/TRUE” as soon as a specific command on the keyboard is pressed by the operator.

The same logic and procedure happen for another digital signal related to the speed value transmission: after the completion of the setup with the mechanical parameters during the main program execution, the two microcontrollers are ready to exchange the reference value of the spindle speed. This crucial operation is executed inside a specific code function, denominated “receive Speed”, and performed by the serial communication available for the Arduino platforms (code 71).

The target value is the angular speed that the motor should reach and maintain during the working cycle to guarantee an optimal result.

Code 71: Enable signal and target speed reception

```
bool receptionState = 0;

int targetSpeed = 0; //target speed of the DC motor [rpm]
float tolerance = 0.0;
int MinimumTargetSpeed = 0;
int MaximumTargetSpeed = 0;
int okSpeed = 0;
bool receiveOK = 0;

int spd = 0;
bool snd = 0;
```

```
int enb = 0;

void setup(){
  Serial3.begin(9600);
}

void loop(){
  startState = digitalRead(startPin);
  secCheckState = digitalRead(secCheckPin);

  enb = receiveEnableStart();

  if(startState == HIGH && secCheckState == LOW && enb == 1){
    digitalWrite(startLed,LOW);
    digitalWrite(pauseLed,LOW);
    digitalWrite(stopLed,HIGH);

    lcdSpeed.clear();
    lcdSpeed.print("Machine ready");
    lcdSpeed.setCursor(0,1);
    lcdSpeed.print("for a process");

    snd = 0; //variable useful for the transmission of the speed

    targetSpeed = 0;
    do{
      startState = digitalRead(startPin); //acquisition of the signal from the
start
      targetSpeed = receiveSpeed();
      if(receiveOK == 2){
        return;
      }else if(startState == LOW){
        Emergency();
        return;
      }
    }while(targetSpeed == 0 && startState == HIGH && receiveOK == 0);
    tolerance = 0.02;
    MinimumTargetSpeed = targetSpeed - tolerance*targetSpeed;
    MaximumTargetSpeed = targetSpeed + tolerance*targetSpeed;

    digitalWrite(startLed,LOW);
    digitalWrite(pauseLed,HIGH);
    digitalWrite(stopLed,HIGH);

    lcdSpeed.clear();
    lcdSpeed.print("Close the window");
    lcdSpeed.setCursor(0,1);
    lcdSpeed.print("to start process");
  }
}

int receiveEnableStart(){
  receiveOK = 0;
  enb = 0;
  do{
    if(Serial3.available()>0) { // Check if data is available to read
      String receivedString = Serial3.readStringUntil('\n'); // Read the incoming
data until newline character
      enb = receivedString.toInt(); // Convert the received string to an integer
for the target speed of the motor
      if(enb == 1){ //check if the speed can be accepted or not, decide a minimum
and a maximum value
        receiveOK = 1;
      }
    }
    startState = digitalRead(startPin);
  }while(startState == HIGH && receiveOK == 0);
```

```
  delay(500);
  return enb;
}

int receiveSpeed(){
  receiveOK = 0;
  spd = 0;
  do{
    if(Serial3.available()>0) { //Check if data is available to read
      String receivedString = Serial3.readStringUntil('\n'); //Read the incoming
data until newline character
      spd = receivedString.toInt(); //Convert the received string to an integer for
the target speed of the motor
      if((spd > 0) && (spd <= 280)){ //check if the speed can be accepted or not,
decide a minimum and a maximum value
        enb_spd = 1;
        Serial3.println(enb_spd);
        receiveOK = 1;
      }
    }
    startState = digitalRead(startPin);
  }while(startState == HIGH && receiveOK == 0);

  lcdSpeed.clear();
  lcdSpeed.print("Speed acquired");
  lcdSpeed.setCursor(0,1);
  lcdSpeed.print("correctly");
  delay(1000);
  return spd;
}
```

After the initialization of the serial communication (in this case the Serial communication n° 3 is used on the pins 14 and 15) in the setup environment, the real transmission can take place. While the serial port is available, the incoming data are saved inside a “string” variable, where each bit is stored and suddenly this set of char variables is converted into an integer data. This cyclic operation is execution as soon as the stop bit “\n” is detected by the function “Serial3.readStringUntil”. the logic can go through if the composed integer, representing the target velocity value, is a feasible value within the minimum and maximum speeds of the motor.

This speed exchange does not occur only in a monodirectional way, but bidirectionally: subsequently, a control signal is sent back to the main Arduino, just to confirm the correctness of the serial communication and its numerical validity.

The final step is the declaration of the successful action to the user, which therefore can mount and secure the workpiece and close the security window of the chuck, because the process is going to start.

An important aspect to underline is the necessary intervention of the human on the preliminary steps of the turning work: this because these procedures require the contact between the man and the machine, and all this must be done in a safe manner and environment to preserve his safety.

4.2.2 Turning work cycle

The execution of the program core starts with the next mandatory command: the enabling of the beginning of the mechanical work by the human operator in a safe condition. Additional controls are related to the closing of the spindle window and the emergency signal (this

command will be analysed later, in the specific function handling the stops of the process caused by an unexpected interruption).

Code 72: Main work functions

```
#define emergencyStopPin 23

int enableState = 0;
bool receiveOK = 0;
int enb_rot = 0;

int i = 0;
int j = 0;
int checkSpeed[100];
int realSpeed = 0;
int velFiltInt = 0;

float error = 0;
float eintegral = 0;
float ederivative = 0;

void setup(){
  pinMode(emergencyStopPin, INPUT_PULLUP);
}

void loop(){
  startState = digitalRead(startPin); //acquisition of the signal from the start
  secCheckState = digitalRead(secCheckPin); //acquisition of the signal from the
securuty window over the spindle

  if(startState == HIGH && secCheckState == LOW && enb == 1){
    lcdSpeed.clear();
    lcdSpeed.print("Close the window");
    lcdSpeed.setCursor(0,1);
    lcdSpeed.print("to start process");

    do{
      secCheckState = digitalRead(secCheckPin);
      startState = digitalRead(startPin);
    }while(secCheckState == LOW && startState == HIGH);
    if(startState == LOW){
      Emergency();
      return;
    }

    enableState = 0;
    do{
      enableState = receiveEnableRotation();
    }while(enableState == 0);
    if(startState == LOW || secCheckState == LOW){
      Emergency();
      return;
    }

    if(startState == HIGH && secCheckState == HIGH && enableState == 1){
      i = 0;
      j = 0;
      realSpeed = 0;
      velFiltInt = 0;
      error = 0;
      eintegral = 0;
      ederivative = 0;
      for(int d=0; d++; d<100){
        checkSpeed[d] = 0;
      }
```

```
      rotationSpindle();
      if(enb_stop == 1){ //normal conclusion of the workcycle
        Stop();
      }else if(startState == LOW || secCheckState == LOW){ //emergency stop of the
workcycle
        Emergency();
        return;
      }else if(emergencyState == HIGH){
        Emergency();
        return;
      }
    }
  }
}

int receiveEnableRotation(){
  receiveOK = 0;
  enb_rot = 0;
  do{
    if(Serial3.available()>0) { // Check if data is available to read
      String receivedString = Serial3.readStringUntil('\n'); // Read the incoming
data until newline character
      enb_rot = receivedString.toInt(); // Convert the received string to an
integer for the target speed of the motor
      if(enb_rot == 1){ //check if the speed can be accepted or not, decide a
minimum and a maximum value
        receiveOK = 1;
      }
    }
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);
  }while(startState == HIGH && secCheckState == HIGH && receiveOK == 0);

  delay(1000);
  return enb_rot;
}
```

At this stage, the core “rotation Spindle” function can be executed (code 73). It mainly contains the reading and analysis of the rotational velocity, the computation of the control signal, which coincides with the input command for the electric motor.

Code 73: Function for the rotation of the spindle

```
bool stopState = 0;

int enb_stop = 0;

void rotationSpindle(){
  enb_stop = 0;

  digitalWrite(startLed, HIGH);
  digitalWrite(pauseLed, HIGH);
  digitalWrite(stopLed, HIGH);

  do{
    startState = digitalRead(startPin);
    secCheckState = digitalRead(secCheckPin);

    if(startState == HIGH && secCheckState == HIGH){

      //commands for the motor and encoder measurements

    }

    if(Serial3.available()>0){ // Check if data is available to read
```

```
      String receivedStop = Serial3.readStringUntil('\n'); // Read the incoming
data until newline character
      enb_stop = receivedStop.toInt(); // Convert the received string to an integer
    }
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}
```

The structure of the "rotation spindle" function is a do-while loop, since the control of the motor occurs in a cyclic way based on the general feedback control loop: at the beginning of the cycle, the reading of the spindle speed is performed, to finish with the motor signal generation. As an outer check, the rotation of the motor can be guarantee if and only if the machine is in the ON state, the security window of the chuck is close protecting the workpiece, and the reading of any stop signal returns a negative output (any stop signal is not sent): as soon as one of these necessary conditions is no longer met, the execution of the function stops, and the spindle must immediately halt its motion.

A remarkable difference must be made regarding the operations that interrupt the work cycle. The normal stop occurs when the mechanical operation ends correctly: at code level the execution exits the "rotation spindle" function and enters another function (detailed later) in other to perform the final stages of the work and finish in a safe manner. On the contrary, a sudden interrupt can be detected by the machine, especially if the machine turns off in a not programmed way or if the security window of the spindle is lifted before the correct ending.

4.2.2.1 Speed reading and filtering

There exist several methods for the speed reading of a motor through an Arduino board. For this project the one chosen is based on the counting of how many times the encoder triggers during a fixed time interval [17] – [18]. This method exploits the functionalities of the Arduino interrupts (code 74).

Code 74: Interrupt generation from the encoder

```
// rotary encoder signals for the motor speed
#define ENCA 2 //signal A
#define ENCB 3 //signal B

void setup(){
  pinMode(ENCA,INPUT_PULLUP);
  pinMode(ENCB,INPUT_PULLUP);

  //interrupt connected to the pin n°2 to trigger when the signal rises
  attachInterrupt(digitalPinToInterrupt(ENCA),readEncoder,RISING);
}
```

Firstly, the global variables that store the number of counts read by the encoder, then the ones useful to store the contribution values throughout the time instants are defined, for the finite approximation approach.

Code 75: Variables for the speed acquisition

```
//global variables to store the number of counts from the encoder
volatile int pos_i = 0;
volatile int increment = 0;
volatile int b = 0;
int pos = 0;
long currT = 0;
float deltaT = 0;
```

```
int posPrev = 0;
long prevT = 0;

float vel = 0;
float velocity = 0;
```

Subsequently, the position of the motor shaft must be acquired. In order to do this, one of the two signals of the encoder must be connected to a pin which can handle the interrupt generation: in this case the signal A is attached to the Arduino digital pin n° 2, and every time the signal rises from 0 V to 5 V, an interrupt is triggered through the command “attach Interrupt” and calls the function “read Encoder” (code 74).

Inside the interrupt service routine, the reading and especially the logical value change of the signal B is acquired (code 76): this leads to determine the direction of the movement, if clockwise or counterclockwise (this distinction is relative to a user reference frame). Also, the encoder signal B is a square wave voltage jumping between 0 V and 5 V.

Then the global variable for the position of the motor is iterated based on the previous evaluation of the rotation direction.

Code 76: Read encoder function

```
void setup(){
  //interrupt connected to the pin n°2 to trigger when the signal rises
  attachInterrupt(digitalPinToInterrupt(ENCA),readEncoder,RISING);
}

void readEncoder(){
  b = digitalRead(ENCB); //acquire the second signal when the first triggers
  increment = 0;
  if(b>0){
    increment = 1; //forward direction (clockwise)
  }else{
    increment = -1; //backward direction (counter-clockwise)
  }
  pos_i = pos_i + increment;
}
```

Back to the loop function, the information about the position of the encoder shaft is stored inside a variable, which is read in an “atomic block” to avoid any potential misread (code 77).

The “ATOMIC BLOCK” is a particular structure of the Arduino programming, and its aim is to prevent the execution of the code inside it by any interrupt from other processes, especially interrupts. For the purpose of the code, it is crucial because inside the atomic block the position of the encoder shaft can be acquired correctly.

To compute the velocity, there is the need of measure the time elapsed between each loop function iteration, starting by saving the present instant with the “micros” command and then determine the delta time by the difference of the present instant and the last one measurement.

Next, compute the difference between the current and the previous encoder count. By taking the ratio between the difference of counts and the time interval, the velocity can be estimated. The unit of measurement will be encoder counts per second.

To obtain the speed with the desired and standard measurement unit, such as rotations per minute, apply the following formula:

$$velocity\ [rpm] = \frac{velocity\ [\frac{counts}{second}]}{600} * 60$$

where 600 is the number of pulses per rotation (resolution of the instrument), 60 to convert seconds to minutes.

Code 77: Speed computation

```
#include <util/atomic.h>

void rotationSpindle(){
  do{
    //read the position in an atomic block to avoid potential misreads
    pos = 0;
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
      pos = pos_i;
    }

    //count the number of pulses within a time unit
    //compute the time between each iteration of the loop function
    currT = micros(); //measure the time elapsed
    deltaT = ((float) (currT - prevT))/1.0e6; //time interval
    vel = (pos - posPrev)/deltaT; //encoder counts per second
    posPrev = pos;
    prevT = currT;

    //convert count/s to RPM and the frequency compensation for the PWM frequency
variation
    velocity = vel/600.0*60.0;
    delay(1); //according to the sampling time
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}
```

During the speed variations, there are frequency oscillations between the switching of the discrete levels. In the frequency domain, the low frequency range measures are coherent and useful, while in the high range only disturbances are present.

A low pass filter is required in order to attenuate the high frequency oscillations of the measurements, yielding to a clearer speed estimation. The filter design has been developed with a simple code written in the Matlab software.

The starting point is the order of the filter function: the first order is enough for the application:

$$H(s) = \frac{\omega_0}{s + \omega_0},$$

where s is the frequency domain variable and ω_0 the cut-off frequency of the function: for the rates before ω_0 the signal is unmodified and preserved, while the contributions above this value are attenuated and almost rejected.

For the project, a cut-off frequency of 10 Hz has been chosen, corresponding to $\omega_0 = 2\pi \cdot (10\ Hz) = 62{,}82$ rad/s. This design is performed in the continuous time domain, but since the speed measurements occur in a discrete time domain, a discretization is required. Exploiting the Matlab command for this operation, the filter function is transformed into an equivalent one using the Tustin discretization method (code 78).

Code 78: Low pass filter design (Matlab)

```
s = tf('s');
ts = 0.001; % sampling time
z = tf('z',ts);

f = 10; % cut-off frequency [Hz]
w0 = 2*pi*f; % cut-off frequency [rad/s]

H = w0/(s+w0);
H_d_tustin = c2d(H, ts,'tustin');
```

Obtaining:

$$H(s) = \frac{62{,}82}{s + 62{,}82} \rightarrow H(z) = \frac{0.03046 * z + 0.03046}{z - 0.9391} = \frac{velocity_{filtered}(k)}{velocity_{raw}(k)}$$

The corresponding Arduino code is written as below (code 79), where also the previous sample of the filtered and the raw signal are involved:

$$vel_{filtered}(k) = 0.9391 * vel_{filt}(k-1) + 0.03046 * vel_{raw}(k) + 0.03046 * vel_{raw}(k-1)$$

Code 79: Low pass filter implementation (Arduino)

```
float velFilt = 0; //filtered output speed at instant k
float velFilt1 = 0; //filtered output speed at instant k-1
float velPrev1 = 0; //unfiltered output speed at instant k-1
int velFiltInt = 0;

void rotationSpindle(){
  do{
    //convert count/s to RPM and the frequency compensation for the PWM frequency
variation
    velocity = vel/600.0*60.0;

    //Low pass first order filter (10 Hz cut-off)
    velFilt = 0.9391*velFilt1 + 0.03046*velocity + 0.03046*velPrev1;
    velPrev1 = velocity;
    velFilt1 = velFilt;

    velFiltInt = (int)velFilt;
    delay(1); //according to the sampling time
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}
```

During the mechanical operation and the motor rotation, its speed is visualized on an LCD display, using this simple piece of code placed inside the cyclic function (code 80).

Code 80: Speed visualization

```
int j = 0;

void rotationSpindle(){
  do{
    velFiltInt = (int)velFilt;

    j++;
    if(j == 49){
      lcdSpeed.clear();
      lcdSpeed.print("Speed: ");
      lcdSpeed.setCursor(0,1);
      lcdSpeed.print(String(velFiltInt) + " rpm");
```

```
      j = 0;
    }
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}
```

The data is shown on the HMI component only every 50-speed evaluation with the counter "j": this choice because the computation of this physical quantity occurs at a frequency way greater than the uploading of the data on the screen.

4.2.2.2 PID controller

The next element implemented on the code, according to the feedback control scheme is the controller block. The aim of this function is to generate a suitable input signal for the electromechanical plant to regulate. Among all possible set of control theories, one widespread method is the PID controller design [19].

Its principle is the computation of an input signal to command a system on the basis of the difference between two physical quantities characteristic of the system under study. The difference, called error, is calculated comparing the reference and the measured signals.

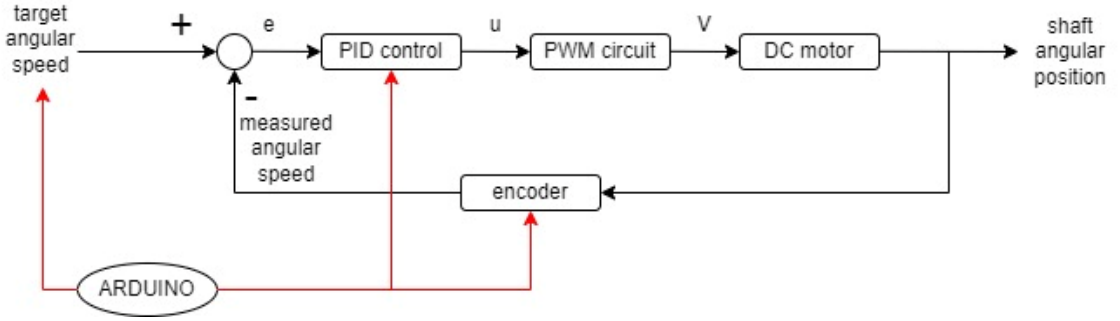


Figure 71: Feedback control scheme for PID controller

In this case the control signal, which is the output of the controller part, is the PWM value to command the switching phase of the MOSFET into the electrical interface circuit.

Mathematically, this signal, generally named "u", aims to reduce as much as possible the error; it is given by the sum of 3 contributions: an additive, an integral and a derivative term, each including the error quantity.

$$Input\ signal\ of\ the\ motor = u = k_P * e + k_D * \frac{de}{dt} + k_I * \int e(t) * dt\ , where$$

$$proportional\ term = P_{out} = k_P * e$$

$$integral\ term = I_{out} = k_I * \int e(t) * dt$$

$$derivative\ term = D_{out} = k_D * \frac{de}{dt}$$

$$and\ e(t) = target\ angular\ speed - measured\ angular\ speed = \omega_{target} - \omega_{measure}$$

The proportional term is the main contribution in the attenuation of the steady state error quantity, while the integral and the derivative terms influence the dynamic of the system response, smoothing its behaviour and stabilizing the response.

Each factor is weighted by a relative non-negative coefficient (degree of freedom of the controller) to tune the response. Typically, the constants K_I is defined as $K_I = K_P/T_I$ and K_D as $K_D = K_P T_D$, where respectively T_I is the integration time (how much time the steady state error is made null) and T_D derivation time (how much time the output dynamics takes to reach the target).

The above formulation of the PID controller is related to the continuous time domain, with the integrator and the derivative operators; since the microcontroller works in a discrete time domain, the formulation must be rewritten in a finite approximation way. For the proportional term, the computation of the error occurs between two discrete instants of time, and it is not a continuous function of the time:

$$e(t) \rightarrow e_{proportional} = e(k) - e(k-1) \; and \; P_{out} = k_P * e_{proportional}$$

The integral term accumulates the error over time, and it is substituted with a discrete summation, according to the formula:

$$\int e(t) * dt \; \rightarrow e_{integral}(k) = e_{integral}(k-1) + e(k) * \Delta t \; and \; I_{out} = \; k_I * e_{integral}$$

And finally, the derivative operator computes how quickly the error is changing, and it is recomposed as:

$$\frac{de}{dt} \rightarrow e_{derivative}(k) = \frac{e(k) - e(k-1)}{\Delta t} \; and \; D_{out} = k_D * e_{derivative}$$

At each iteration of the loop function, the input signal is evaluated on the basis of the previous instant of time speed value. The error is computed and stored inside the relative variable; therefore, all the 3 contributions are calculated using the above formulations, to finish with the PID terms including the coefficients.

Remarkable focus must be spent for the numerical quantity of the integral term: since it accumulates the error during the process (integrator and discrete sum effect), it might assume very high values, leading to a saturation of the response and compromising the PID control action. A possible solution to this problem is to bound its contribution, that is the anti-wind-up technique. The two limits chosen are the minimum and the maximum values that the PWM signal can assume, respectively 0 and 255.

A possible misbehaviour regards also the time measurements useful to acquire the rotational speed. Especially at the beginning of the rotation, the delta time detected is too big and this leads to the saturation of the integral contribution. In order to solve this problem, a further experimental anti-wind-up approach has been applied: in addition to the limit of the saturation of the integral contribution, also a check on the delta time is performed. When this quantity is greater than a comparison sample (set to an experimental 0,1 seconds), the PID output is just the proportional term just to avoid the misleading integral contribution, otherwise all the 3 terms are considered and summed.

Finally, the update/storage of the present error variable, which will become the previous error value in the next loop iteration.

Code 81: PID control function

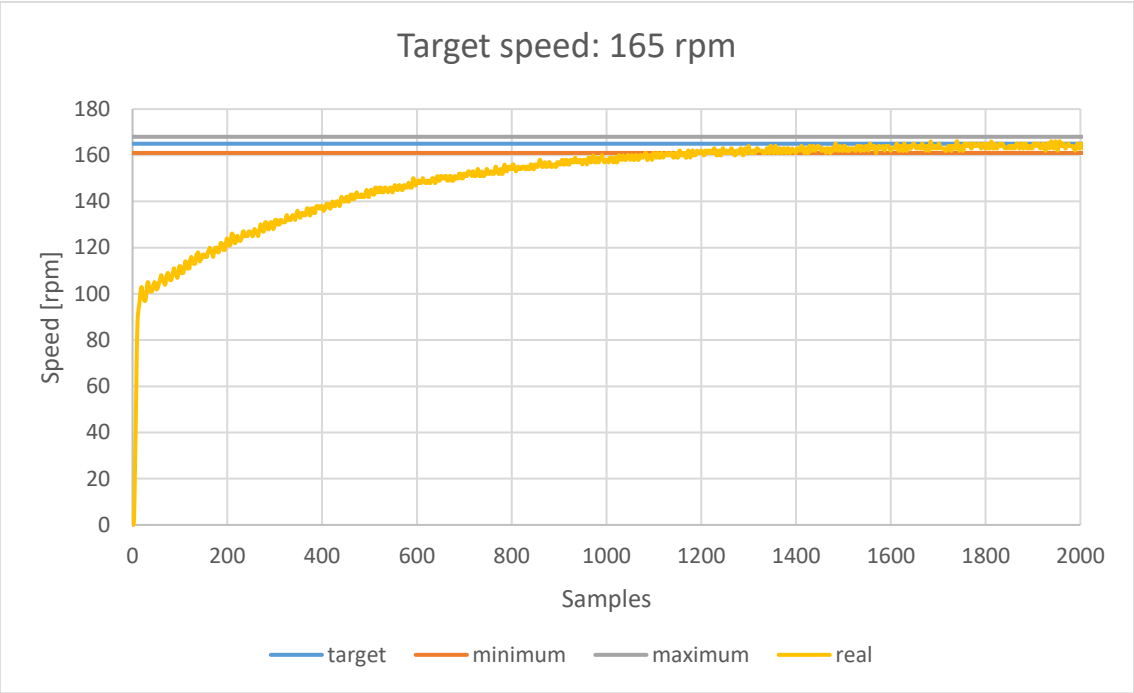
```
float kp = 0; //proportional constant
```

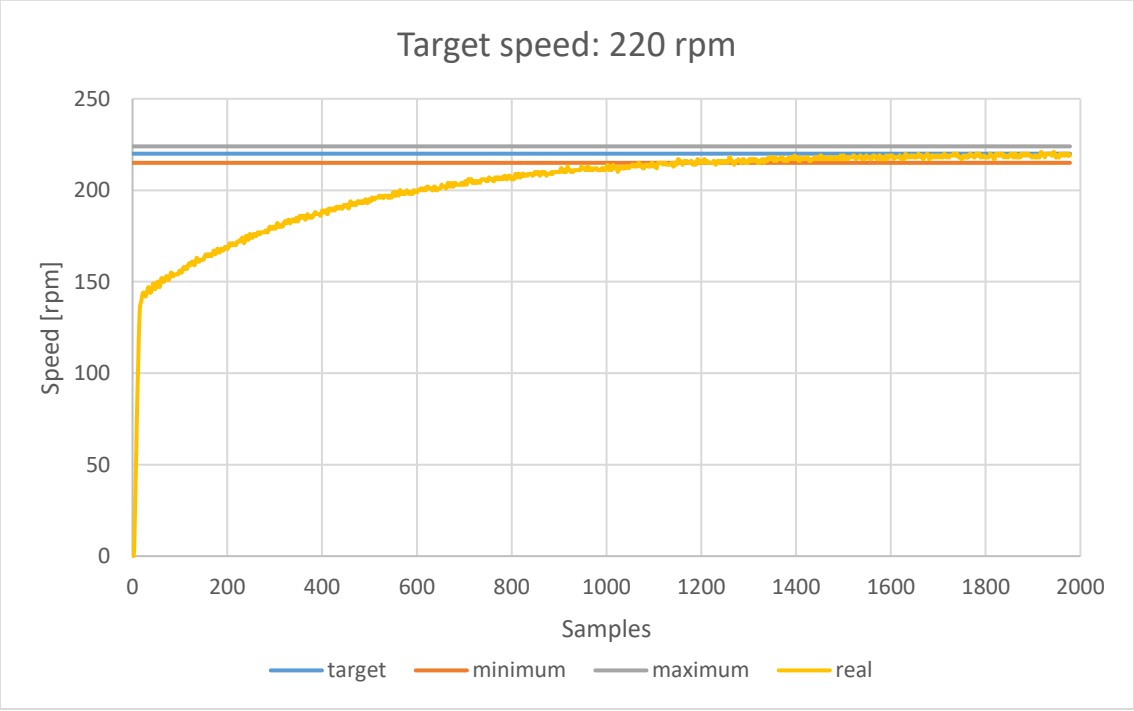
```
float ki = 0; //integrative constant
float kd = 0; //derivative constant

float error = 0;
float eintegral = 0;
float ederivative = 0;
float lastError = 0;

float LowerLimit_Iout = 0;
float UpperLimit_Iout = 255;

float Pout = 0;
float Iout = 0;
float Dout = 0;
float u = 0;

float sampleTime = 0.1; //[seconds]

void rotationSpindle(){
  do{
    //compute the control signal u
    kp = 2.0;
    ki = 0.13;
    kd = 12;

    u = PIDcontroller(targetSpeed, kp, ki, kd);
    delay(1); //according to the sampling time
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}

float PIDcontroller(int target, float KP, float KI, float KD){
  error = (float) (targetSpeed - velFiltInt);
  eintegral = eintegral + error*deltaT;
  ederivative = (error - lastError)/deltaT;

  Pout = KP*error;
  Iout = KI*eintegral;

  if(Iout < LowerLimit_Iout){
    Iout = LowerLimit_Iout;
  }else if(Iout > UpperLimit_Iout){
    Iout = UpperLimit_Iout;
  }

  Dout = *ederivative;

  if(deltaT <= sampleTime){
    u = Pout + Iout; + Dout;
  }else if(deltaT > sampleTime){
    u = Pout;
    eintegral = 0;
  }

  lastError = error;
  return u;
}
```

The tuning of the 3 parameters has been made manually applying a trial-and-error approach: the final values used inside the code are $K_P = 2.0$ $K_I = 0.13$ and $K_D = 12$.

Firstly, both K_I and K_D have been set to 0 and modified only K_P until the steady state error assumes an acceptable value. Then K_I is increased in order to decrease as much as possible the error: the value must not be augmented too much to avoid instability. Finally, the K_D term is adjusted to reduce the amplitude of the oscillations of the output variable.

Thanks to the action of the controller, the speed increases gradually over time and, after around 20-30 seconds, it reaches the steady state condition, represented by the target speed.





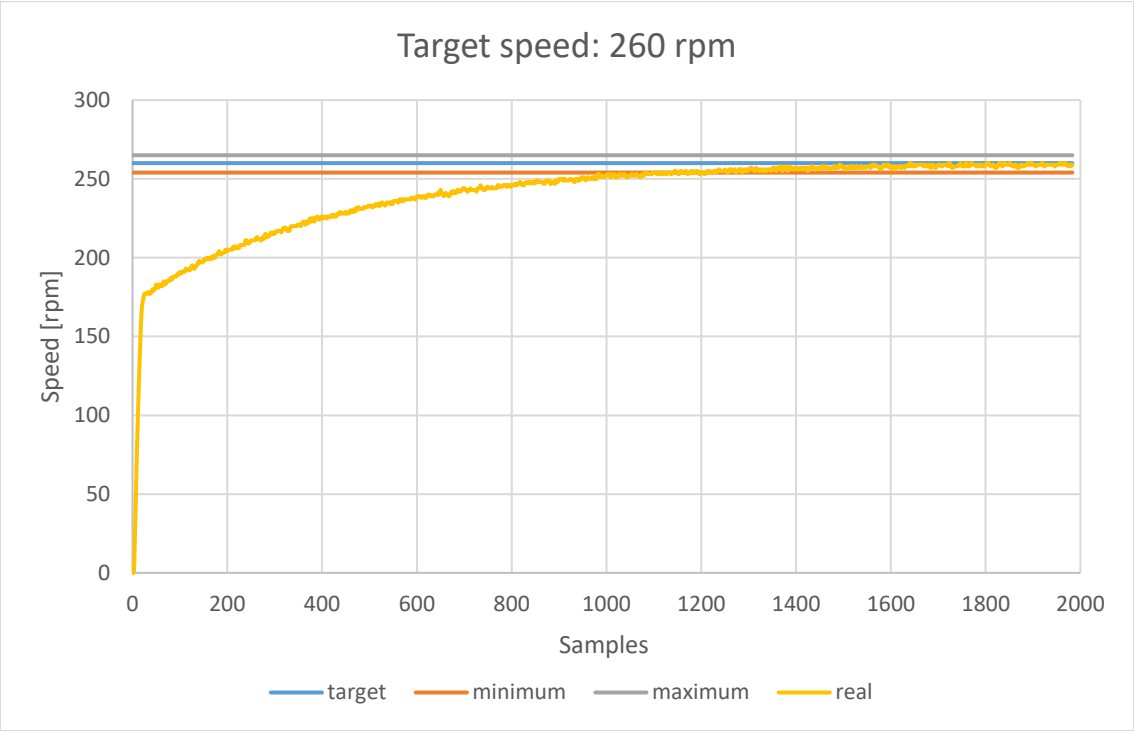


Figure 72: Examples of speed dynamic with 3 target values

4.2.2.3 Motor command

The just computed control signal value must be injected inside the motor function exploiting the PWM connections of the Arduino board (code 82). It controls the switching phase of the MOSFET.

Since the variable "u" can take on any value, a limitation of its quantity is required: specially the value "u" cannot exceed the maximum admissible value of 255. To accomplish this check, firstly only the integer part and the absolute value of the control signal "u" is saved, with the function "(int) fabs(u)", then perform the real limitation with the if structure.

Code 82: Motor command function

```
float u = 0;
#define pwmPin 6

void setup(){
  pinMode(pwmPin,OUTPUT);
  setPwmFrequency(pwmPin,1); //set the pin and the divisor on the timer for the PWM
signal
}

void rotationSpindle(){
  do{
    setMotor(u,pwmPin);
    delay(1); //according to the sampling time
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}

//function to drive the motor
void setMotor(float input, int PinMotor){
  int pwmVal = (int) fabs(u);
  if(pwmVal > 255){
    pwmVal = 255;
  }
```

```
  analogWrite(PinMotor,pwmVal);
}
```

4.2.2.4 Real speed communication

Once the motor and the spindle start rotating, a certain amount of time is dedicated to the reaching of the steady state value for the angular speed; the steady state condition is symbolised by the reference speed.

As soon as the signal response reaches an almost constant quantity around the target one (a tolerance band is planned with bounds of the ±2%), this information is transmitted with the serial communication to the main Arduino (code 83): this is the enabling signal for the start of the mechanical operation.

A certain tolerance stripe must be designed because the motor cannot rotate precisely to a constant speed in time, due to the physical limitations of the controller, the Arduino resolution computing the control input signal and the non-perfect construction of the motor wirings.

Before of the real transmission of the speed information, a certain number of data is collected inside an array, meaning that the motor can maintain, and it stabilizes around the target velocity constantly in time.

Code 83: Real speed value communication

```
int i = 0;
int realSpeed = 0;

void rotationSpindle(){
  do{
    //check of the speed to send to the Arduino the real spindle speed
   if(velFiltInt >= MinimumTargetSpeed && velFiltInt <= MaximumTargetSpeed && snd
== 0){
        checkSpeed[i] = velFiltInt;
        i++;
        if(i == 99){
          realSpeed = velFiltInt;
          snd = 1;
          Serial.println("");
          Serial.print("\t \t");
          Serial.println(realSpeed);
          Serial.println("");
          sendSpeed();
        }
      }else if((velFiltInt <= MinimumTargetSpeed || velFiltInt >=
MaximumTargetSpeed) && snd == 0){
        for(int d=0; d++; d<100){
          checkSpeed[d] = 0;
        }
        i = 0;
      }
    delay(1); //according to the sampling time
  }while(startState == HIGH && secCheckState == HIGH && enb_stop == 0);
}
```

Inside the code, an additional action has been implemented: if a speed value falls outside the tolerance band during the collection of the data, and so before the serial transmission, the acquisition halts; a new collection re-starts at the first admissible new value.

4.2.3 Stop function

As mentioned before, the motion of the motor stops immediately if the machine turns off or if the security window open: these two behaviours can be considered as stops in an emergency or, however, halts caused by the external intervention.

Considering the normal execution of the work, so without any sudden interruption, the motor can stop when a specific signal is generated by the main code and read by this Arduino board (code 84). Once the work on the product finishes correctly, the stop variable switches to the “HIGH” state. Immediately the code exits the loop function and enters the stop mode, where the motor input variable is forced to be null. Clear all the unnecessary variables are cleared.

Code 84: Stop function

```
void loop(){
  if(secCheckState == HIGH && enableState == HIGH && emergencyState == LOW){
    rotationSpindle();
    if(enb_stop == 1){ //normal conclusion of the workcycle
      Stop();
    }else if(startState == LOW || secCheckState == LOW){ //emergency stop of the
workcycle
      Emergency();
      return;
    }
  }else if(emergencyState == HIGH){
    Emergency();
  }
}

//stop the rotation of the spindle when the process of turning is finished
void Stop(){
  targetSpeed = 0;
  u = 0;
  setMotor(u,pwmPin); //stop the motor

  digitalWrite(emergencyLed,LOW);
  digitalWrite(stopLed,HIGH);
  digitalWrite(pauseLed,LOW);
  digitalWrite(startLed,LOW);

  lcdSpeed.clear();
  lcdSpeed.home();
  lcdSpeed.print("The process");
  lcdSpeed.setCursor(0,1);
  lcdSpeed.print("is completed");

  realSpeed = 0;
  velFiltInt = 0;
  for(int d=0; d++; d<100){
    checkSpeed[d] = 0;
  }

  do{
    secCheckState = digitalRead(secCheckPin);
  }while(startState == HIGH && secCheckState == HIGH);
}
```

Following the main program sequence, only after the complete retraction of the carriage to the home position, the window can be opened by the user safely.

The machine returns to the starting point and a new work cycle can begin.

4.2.4 Emergency stop

An additional functionality needs to be added to the code when the process is subject to a sudden stop: this scenario can be compared to an emergency situation.

The program enters the emergency when a specific signal is detected. Again, the motor must stop immediately to solve the possible misbehaviour safely.

The logic can go back to the starting point only after the change of the logical value of the emergency signal.

Code 85: Emergency function

```
void loop(){
  if(secCheckState == HIGH && enableState == HIGH && emergencyState == LOW){
    i = 0;
    j = 0;
    realSpeed = 0;
    velFiltInt = 0;
    for(int d=0; d++; d<100){
      checkSpeed[d] = 0;
    }
    rotationSpindle();
   if(enb_stop == 1){ //normal conclusion of the workcycle
      Stop();
    }else if(startState == LOW || secCheckState == LOW){ //emergency stop of the
workcycle
      Emergency();
      return;
    }
  }else if(emergencyState == HIGH){
    Emergency();
  }
}

void Emergency(){ //stop the motor, switch on the emergency/fault led
  targetSpeed = 0;
  u = 0;
  setMotor(u,pwmPin); //stop the motor

  digitalWrite(emergencyLed,HIGH);
  digitalWrite(startLed,LOW);
  digitalWrite(pauseLed,LOW);
  digitalWrite(stopLed,LOW);

  lcdSpeed.clear();
  lcdSpeed.home();
  lcdSpeed.print("EMERGENCY");
  lcdSpeed.setCursor(0,1);
  lcdSpeed.print("STOP");

  realSpeed = 0;
  velFiltInt = 0;
  for(int d=0; d++; d<100){
    checkSpeed[d] = 0;
  }

  do{
    emergencyState = digitalRead(emergencyStopPin);
  }while(emergencyState == HIGH);
}
```

4.3 Design of a boost converter

Exploiting the capabilities of the electrical circuit analysed before, the maximum angular speed reachable by the workpiece is around 280 rpm: this network has been useful to test the Pulse Width Modulation technique with the Arduino control board and the related code. In the world of the turning operations, the speed can reach greater values: a more powerful circuit must be designed. A possibility is to employ a particular topology of electrical converters, caller boost converter, able to generate the right amount of power for the motor.

It is specifically a DC-DC boost converter or step-up circuit. Its main working principle is to give a variable output voltage, which is increased with respect to a fixed and lower input voltage.

This electrical network will be interposed between the power generator and the motor itself and controlled by the Arduino microcontroller.

This design is a preliminary study, just to present a possible solution to improve the power stage, accompanied by the theoretical design and the choice of the theoretical electrical components to build it.

The voltage at the input stage is 15 V, which is generated constantly by a power supply: this component converts the AC voltage 230 V at 50 Hz of the electrical line into a constant voltage of 15 V.

This value of voltage is too low to be directly applied to the motor because the rotational speed achievable would be too slow for a lathe process, so an increase of voltage is required: a step-up converter is designed for this purpose. On the other hand, it has been chosen since it is able to generate a lower speed than the previous circuit, and so a compromise is required. The power supply must provide enough current to accomplish the task, based on the theoretical calculations and simulations.

Later, a theoretical design and a simulation implementation have been performed to identify the more appropriate sizes of the electronic components, just to test its effectiveness.

4.3.1 Theoretical design and simulation with LTSpice software

The rotational speeds of the spindle must be set for the different turning processes, and this means that a variable voltage and current must be applied to the motor. As anticipated before, the boost converter circuit is able to provide higher voltages starting from a fixed input voltage; exploiting the capabilities of the Arduino board to provide a control signal which is a square wave with variable duty cycle (PWM technique), the output voltage can be adjusted to the desired value useful for the working application.

According to the PMDC motor specifications, its electrical power is 150 W, while the maximum voltage and current that can be applied are 230 V and 0.9 A, but during the design a lower voltage is considered arising from the calculations.

The power is the starting point of the theoretical design, since according to the conservation of energy law, the input power provided by the power supply and the output power directed to the motor must be equal (this can be stated with the assumption of no losses in the circuit). The consequence of this conservation is that, if the output voltage must be greater than the input one, the current injected into the circuit will be greater than the load current obtained:

$$Input\ power\ P_{in} = Output\ power\ P_{out} \rightarrow V_{in} < V_{out}\ and\ I_{in} > I_{out}$$

Boost converters are a type of DC-DC switching converter that efficiently increase or step-up the input voltage to a higher output value [20].

The general layout of the circuit is Fig. 73. The key components required are an inductor, a diode, two capacitors and a switch (transistor MOSFET), where its opening-closing frequency is set by the Arduino PWM connection.

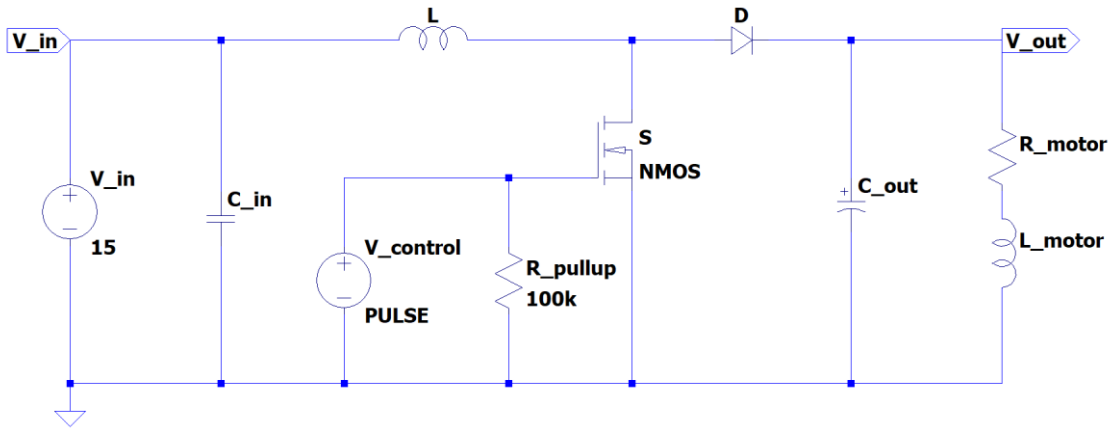


Figure 73: General scheme of a DC-DC boost converter

By storing the magnetic energy in an inductor during the switch-on phase and releasing it to the load (the motor in this case) during the switch-off phase, this voltage conversion is made possible.

The operation of the boost converter is based on two stages:

1. Switch-on period (switch closed/on and diode open): during this stage, the input voltage is applied across the inductor, causing the current through the inductor to increase linearly from its minimum to maximum value. The energy stored in the inductor builds up, and the diode is reverse-biased by the voltage at the load that is supplied with energy from the capacitor, preventing current flowing to the load. The inductor current is expressed as:

$$\Delta I_L = \frac{V_{IN}}{L} * t_{ON}$$

Where ΔI_L is the change in inductor current, L is the inductance, and t_{ON} is the duration of the switch-period.

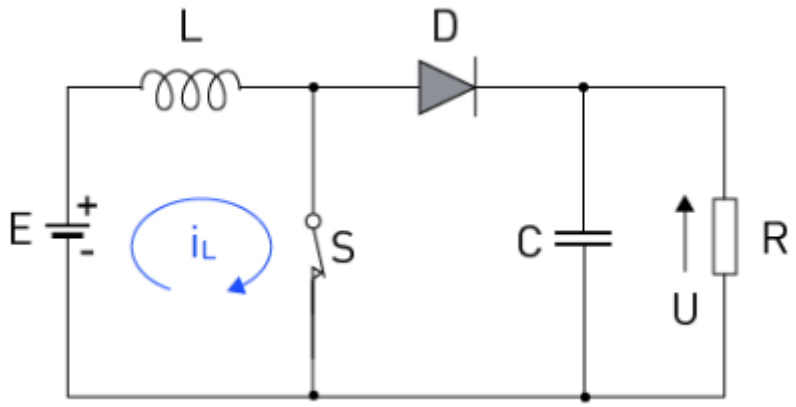


Figure 74: Boost converter: interval t_ON

2. Switch-off period (switch open/off and diode closed): when the transistor opens, the inductor current must continue to flow. This forces the diode to become forward-biased, and the inductor releases it stored energy to the load and the output capacitor. During this period, the voltage across the inductor changes direction and becomes equal to the difference between the output voltage and the input voltage. The inductor current decreases linearly as the energy is transferred to the load, and the equation of the current becomes:

$$\Delta I_L = \frac{V_{OUT} - V_{IN}}{L} * t_{OFF}$$

Where t_{OFF} is the duration of the switch-off period. The inductor current decreases from its maximum to its minimum value.

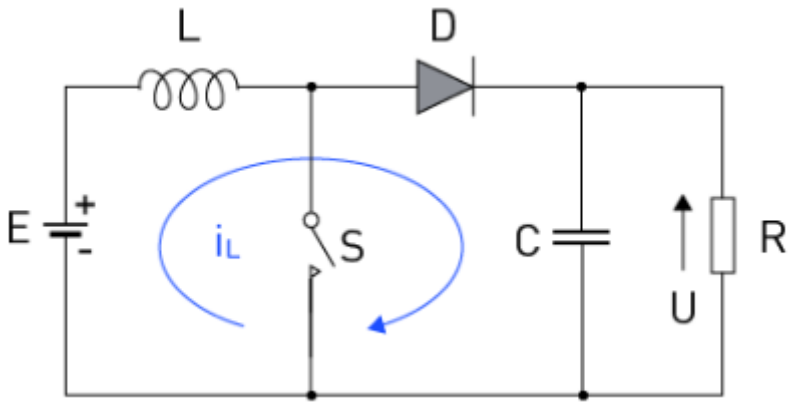


Figure 75: Boost converter: interval t_OFF

In the steady state the average voltage value on the inductor is equal to zero; by equating the inductor current equations, the voltage conversion relationship for the boost converter becomes:

$$V_{OUT} = \frac{V_{IN}}{(1 - D)}$$

Where D is the duty cycle, defined as the ratio of the switch-on time (t_{ON}) to the total switching period T, or:

$$D = \frac{t_{ON}}{t_{ON} + t_{OFF}}.$$

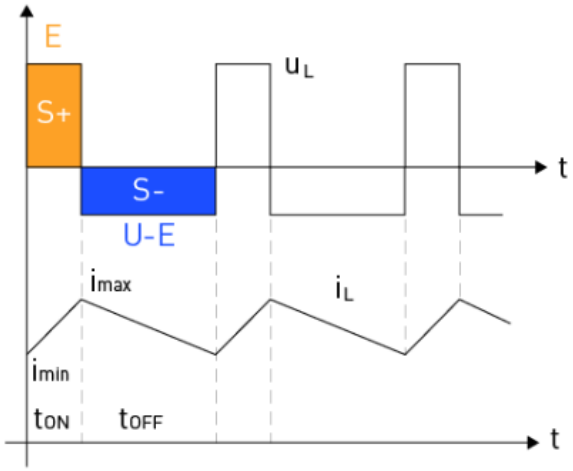


Figure 76: Inductor voltage and current in a time graph

The voltage at the load will be equal to the DC source's voltage if D=0, and it will increase in the active part of the period (t_{ON}). On the other hand, the voltage will be infinitely large if D=1, which is not possible because the duty cycle equal to 1 implies that $t_{ON} = T$or $t_{OFF} = 0$, which further implied that the transistor is never turned off and thus no energy transfer will occur from the DC source to the load, resulting in the voltage being equal to zero.

So, the key components are the following:

- Inductor: stores and releases energy throughout the switching cycles. Its major job is to preserve energy storage during the conversion while controlling current flow.
- Switch: by alternating disconnecting and connecting the inductor to the load, the switch, translated in this case into a MOSFET, plays a crucial role in managing the energy transfer between the input and output.
- Diode: during the switch-off time, the diode only permits current to flow in one direction in the circuit – toward the output. When the switch is open, it stops the current from returning to the input side.
- Output capacitor: it is in charge of stabilizing the voltage across the load and smoothing the output voltage. It guarantees a constant output voltage for the load and filters out high-frequency voltage ripples.
- Load: symbolizes the electrical load that is linked and powered by the boost converter, which is built to give it the necessary voltage and current. In this project the load is the DC motor with the model of its resistance and inductance.

In the analysis and design of boost converters, it is crucial to consider the two primary conduction modes: continuous conduction mode (CCM) and discontinuous conduction mode (DCM). Both modes differ in the behaviour of inductor current, which affects the converter's performance, efficiency, and design criteria.

- Continuous Conduction Mode (CCM): In CCM, the inductor current remains positive and never reaches zero throughout the switching cycle. The inductor current grows during the switch-on phase, storing energy in its magnetic field. The inductor current drops as

it transfers its stored energy to the output when the switch is switched off. The current waveform in CCM has a triangle shape, and the load current is equal to the average value of the current. Lower peak currents and output voltage ripple are produced by CCM operation, boosting efficiency and reducing strain on the converter's parts. To sustain continuous current flow, greater inductor values are also necessary, which could result in an increase in the converter's size and price.

- Discontinuous Conduction Mode (DCM): In DCM, during a portion of the switching cycle, the inductor current drops to zero, signifying that the energy transfer to the output is finished before the start of the subsequent switch-on period. There is a zero-current interval between the switch-off and the subsequent switch-on periods, giving the inductor current waveform in DCM a trapezoidal shape. Smaller inductor values are possible with DCM operation, which results in compact and affordable converter designs. In contrast to CCM, it also leads to larger peak currents, increased output voltage ripple, and lower efficiency. DCM is more frequently used in applications with a wide range of load circumstances or light loads.

At this stage, the design focuses on the dimensioning of the components of the circuit, considering the continuous conduction mode.

The first passage is to determine the application's requirements translated into the input and output voltages and the currents. The converter's needed voltage conversion ratio, duty cycle and power handling capacity will be determined by these specifications.

$$V_{IN} = 15\ V\ and\ V_{OUT} = \sqrt{P_{out} * Z_{motor}} = \sqrt{150 * 28} = 65\ V$$

Where Z_{motor} is the impedance of the motor.

The corresponding duty cycle valid for the CCM operation is:

$$Duty\ cycle = D = 1 - \frac{V_{in}}{V_{out}} = 77\%$$

The output current is set to 0,9 A and the input current is determined by the formula acquired by the literature [21]:

$$I_{IN} = \frac{V_{IN}}{(1 - D^2) * Z_{motor}} = \frac{15}{(1 - 0{,}77^2) * 28} = 1{,}31\ A$$

This value is important for selecting the appropriate switching frequency and inductor value.

The converter's efficiency, transient response, and size of the passive components are all impacted by the switching frequency. Compact designs are made possible by the lower inductor and capacitor values at higher switching frequencies. The larger switching losses and decreased efficiency could result from the higher switching frequency. Size and efficiency must be compromised while choosing the switching frequency.

The inductor value is crucial for maintaining the desired conduction mode and ensuring a stable operation. Its formula for the CCM is:

$$L = \frac{V_{in} * D}{\Delta I_L * f}$$

The current ripple on the inductor, ΔI_L, is evaluated as:

$$\Delta I_L = \frac{V_{IN}}{L} * t_{ON} = \frac{V_{IN}}{L} * d * T = \frac{V_{IN} * d}{L * f} \rightarrow L = \frac{V_{in} * D}{\Delta I_L * f}$$

Where ΔI_L is the peak-to-peak inductor current ripple. A smaller inductor value results in a higher current ripple, which may affect the output voltage ripple and converter efficiency.

The output capacitor C_{OUT} helps to filter the output voltage ripple and ensure a stable output. Its value can be determined as:

$$C_{OUT} = \frac{I_{OUT} * D}{\Delta V_{out} * f}$$

The output capacitor C_{OUT} controls converter's stability by filtering output voltage sag. The intended output voltage ripple ΔV_{out}, the load current, the duty cycle D, and the switching frequency must be taken into account to obtain the proper value for C_{OUT}. The output voltage ripple ΔV_{out} is mainly due to the inductor current ripple ΔI_L charging and discharging the output capacitor during the switching cycle. In a boost converter, the inductor current ripple ΔI_L flows through the output capacitor during the off-time of the switch t_{OFF}, when the diode is conducting.

Firstly, consider the capacitor current during the off-time. As the inductor current is approximately constant during this period, approximate the capacitor current:

$$I_C \approx \Delta I_L = I_L * (1 - D) = 1{,}31 * (1 - 0{,}77) = 0{,}3\ A$$

And the inductance:

$$L = \frac{V_{in} * D}{\Delta I_L * f} = \frac{15 * 0{,}85}{0{,}29 * 31250} = 1{,}22\ mH$$

Consider the relationship between the current and the capacitance in the time domain:

$$I_C = C * \frac{dV}{dt}$$

Where I_C is the capacitor current, C the capacitance value and $\frac{dV}{dt}$ the derivative of the voltage across the capacitor in time.

Get:

$$\Delta I_L * (1 - D) \approx C_{OUT} * \frac{\Delta V_{OUT}}{dt}$$

Where T is the switching period (1/f), and ΔV_{OUT} is the desired output voltage ripple, assumed equal to 0,1 V.

Rearranging the equation to solve for C_{OUT}:

$$C_{OUT} \approx \frac{\Delta I_{OUT} * (1 - D) * T}{\Delta V_{OUT}}$$

The inductor current ripple is directly proportional to the output current and the duty cycle. A higher output current will lead to a larger inductor current ripple. Thus, the equation for the output capacitance can be rewritten as:

$$C_{OUT} \approx \frac{I_{OUT} * D * T}{\Delta V_{OUT}} = \frac{D}{Z_{motor} * \frac{\Delta V_{OUT}}{V_{OUT}} * f} = \frac{0{,}77}{28 * \frac{0{,}1}{65} * 31250} = 878\ \mu C$$

Where ΔV_{OUT} is the desired peak-to-peak output voltage ripple, a larger capacitor value will reduce the output voltage ripple but may increase the size and cost of the converter.

The voltage, current, conduction, and switching losses of the switch (often a MOSFET) and the diode must be taken into consideration while choosing them. While the current rating should surpass the peak current in the converter, the voltage rating should be greater than the maximum voltage across the switch and diode when it is operating.

Finally, an input capacitance is placed in parallel with the power supply to stabilize the input voltage due to the peak current requirement of a switching power supply. It has been chosen equal to 100 nC.

In a boost converter, efficiency is a crucial parameter determining how effectively it transfers power from input to output. The efficiency (η) of a boost converter can be expressed as the ratio of output power (Pout) to input power (Pin):

$$\eta = \frac{P_{OUT}}{P_{IN}}$$

The losses in a boost converter can be attributed to several factors:

- Conduction Losses: The resistance of circuit elements like the inductor, switch (MOSFET), and diode causes conduction losses. These losses are directly related to the resistance of the components and the current that flows through them. By choosing parts with lower resistances and improving the converter's design for low current flow, conduction losses can be reduced.
- Switching Losses: When a MOSFET switches between its on and off states, switching losses occur. During these transitions, the switch is experiencing both voltage and current, which results in power loss. The MOSFET's on-state resistance, drain-to-source voltage, and switching frequency all influence the overall switching loss. A MOSFET with low on-state resistance and quick switching periods should be used to reduce switching losses.
- Diode Reverse Recovery Losses: A diode experiences reverse recovery as it transitions from the conducting state to the blocking state. Power loss occurs as a result of the diode briefly conducting current in the opposite direction during this process. A Schottky diode or other diode with a short reverse recovery time should be chosen to reduce these losses.
- Magnetic Core Losses: The alternating magnetic field inside the inductor core results in energy dissipation in the form of heat and causes inductor core losses. The core material, operation frequency, and magnetic flux density all affect these losses. To reduce core losses, an appropriate core material should be selected, and the inductor design should be optimized to minimize the magnetic flux density.
- Capacitor Losses: Equivalent series resistance (ESR) of the output capacitor is the principal cause of capacitor losses. These losses lead to the production of heat and a

decline in general efficiency. These losses can be reduced by using capacitors with a low ESR.

The resultant circuit is depicted in Fig. 77 and two tests with two different duty cycle values (Fig. 78 and Fig. 79):

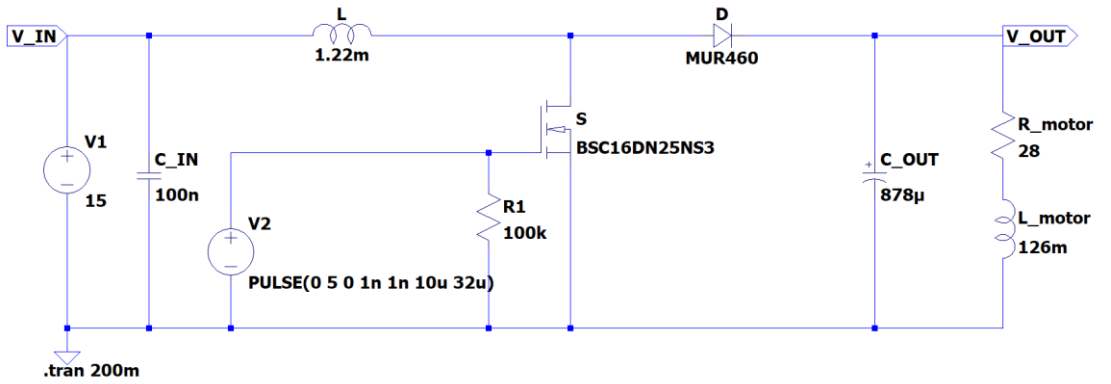


Figure 77: Boost converter circuit with values

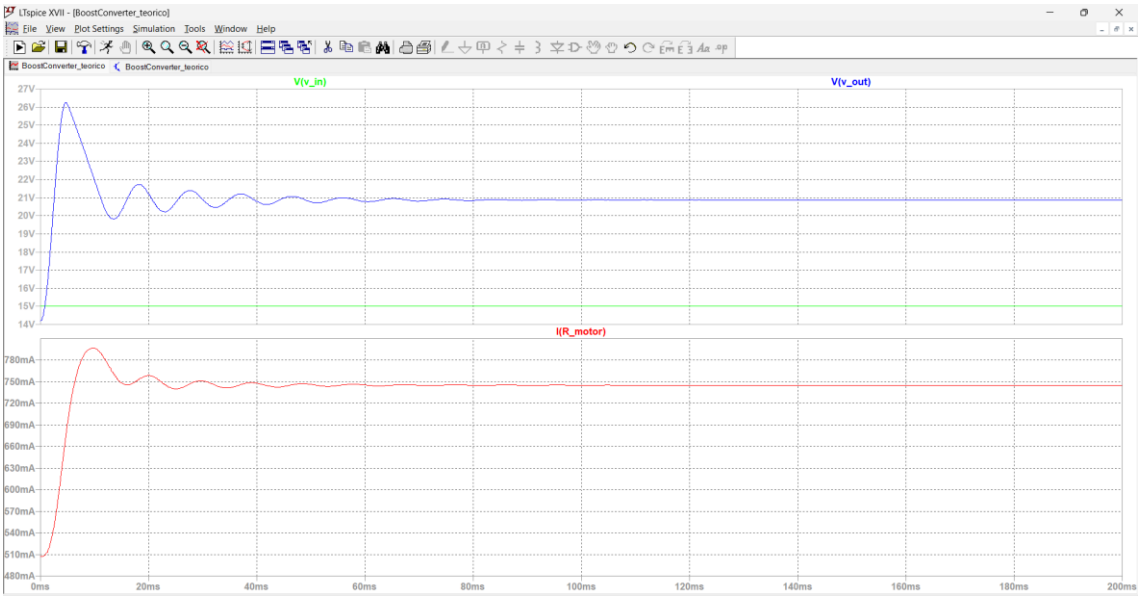


Figure 78: Voltage and current of the motor with 31,25% duty cycle

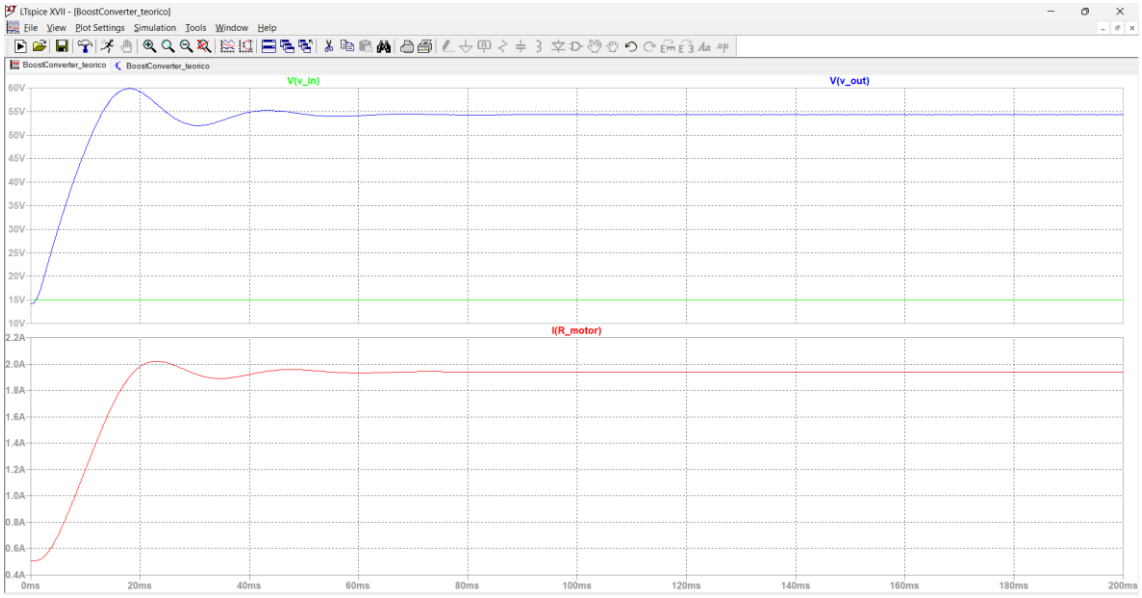


Figure 79: Voltage and current of the motor with 75% duty cycle

Through the simulation software, the goal of increasing the voltage has been achieved, but some discrepancies occur regarding its numerical value and the current flowing: with a more accurate components choice, these errors can be decreased, considering the non-ideal behaviour of the simulated elements as well.

5. Mechanical design

The last section of the project regards the design and realization of the components added to the lathe machine in order to allocate all the additional components. The design of these features has been developed without massive modifications of the original structure of the machine.

Focus has been paid to the manufacturing of the supports and joints of the encoder and stepper motors linked to the two lead screws and the supports of the ultrasonic sensor and the HMI features.

The mechanical design of the products has been done on a 3D modelling software, SolidWorks: starting from the measure of the quotes of the manual lathe, a digital model has been realized, in order to design the additional elements.

In the Fig. 80 the overall model is depicted with the labels of the new accessories.

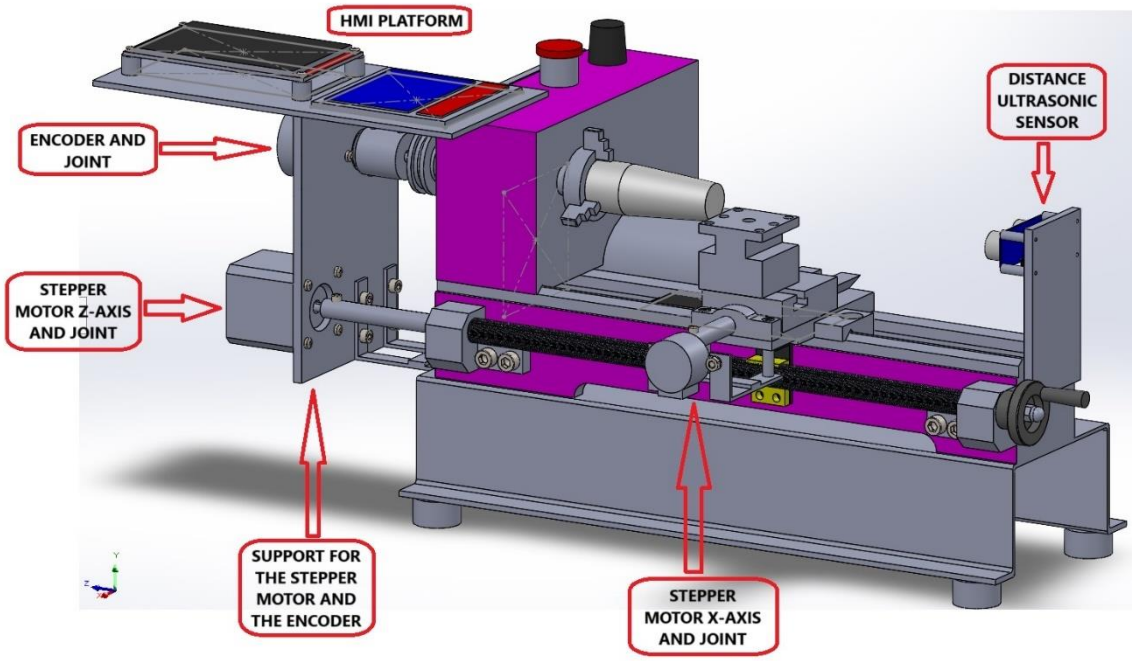


Figure 80: 3D CAD model of the lathe machine

The following figures depict the exploded views of the component's assemblies including the specifications of the commercial tools from the SolidWorks toolbox.

1. Support with the longitudinal stepper motor and the encoder (Fig. 81): 4 ISO 7045 M3x10 screws for the stepper motor, 3 ISO 7045 M3x10 screws for the encoder and 4 ISO 4762 M4x12 screws and 4 ISO 4032 M4 (hex) screws for the clamps of the support.

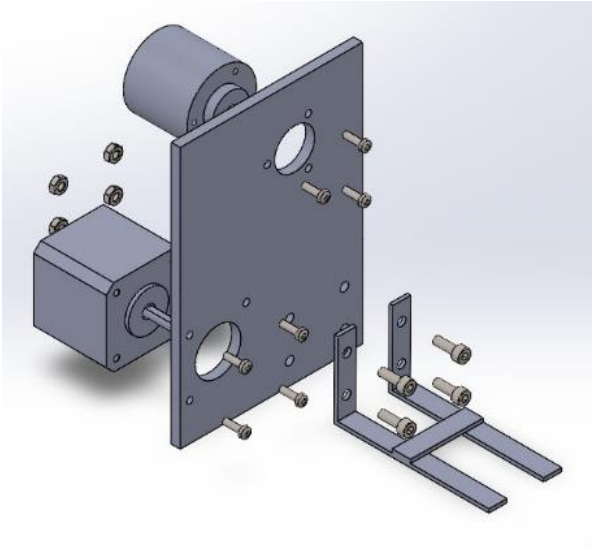


Figure 81: Support with the encoder and stepper motor Z-axis

2. Longitudinal stepper motor with the joint (Fig. 82). The joint of the stepper motor is linked to the side edge of the lead screw providing the longitudinal motion of the carriage. Realization of a cavity to allocate the tab of the lead screw. On the other side the joint is tightened to the shaft of the stepper motor via a screw grain (ISO 4026 M5x6), exploiting the flat surface of the shaft.

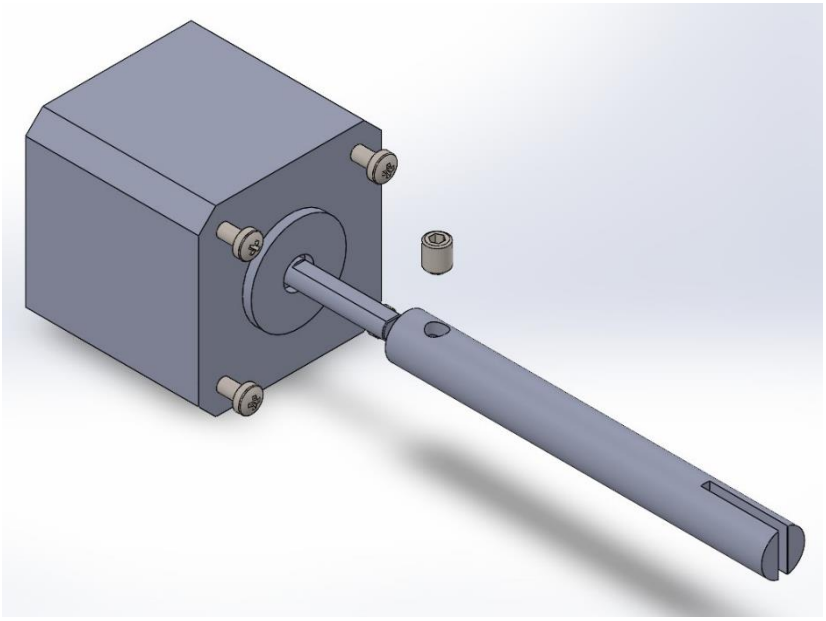


Figure 82: Stepper motor Z-axis with the joint

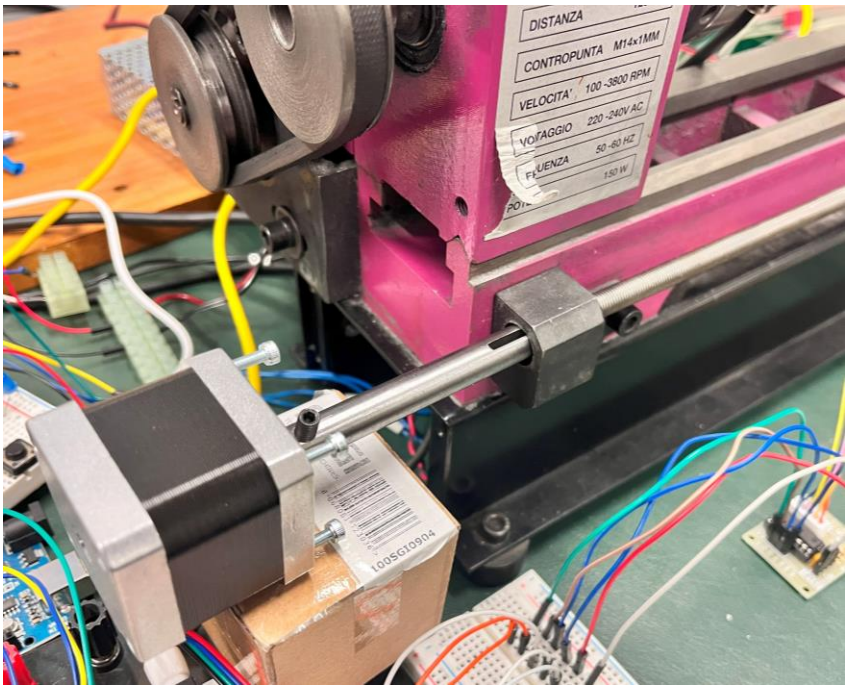


Figure 83: Mounting of the stepper motor on the lathe machine

3. Encoder with the joint (Fig. 84). The mechanical joint of the encoder is split into two pieces: one flexible joint and a multidiameter shaft. The latter is inserted inside a hole of the spindle pulley, and the flexible joint links this shaft and the encoder output shaft with two ISO 4026 M4x8 screw grains.

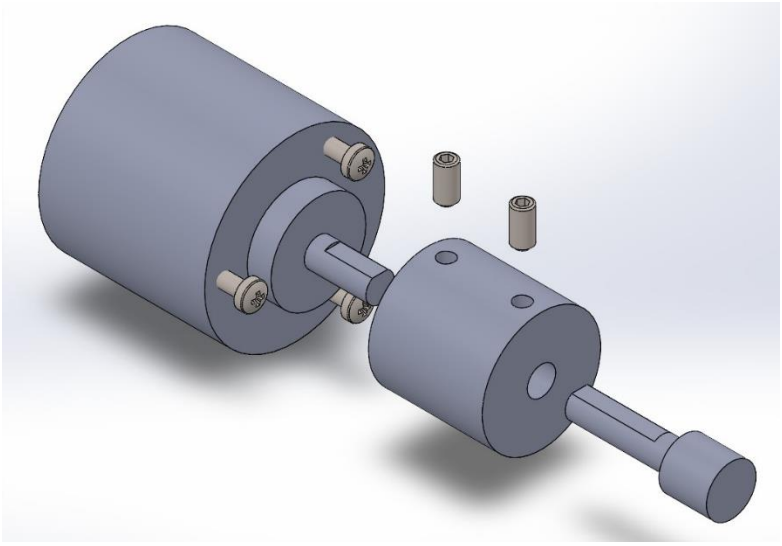


Figure 84: Encoder with the joint

4. Stepper motor X with the joint (Fig. 85). The stepper motor responsible for the transversal motion of the cross slide is held up with two clamps to the carriage (2 ISO 4762 M4x6 screws and 2 ISO 4032 M4 hex nut for the clamps; 2 ISO 4762 M3x30 screws and 2 ISO 4032 M3 hex nut for the clamps of the support on the carriage side). The output shaft of the motor is linked to the lead screw of the carriage with a joint; on one side the connection is performed with an ISO 4026 M5x6 screw grain and on the other through an internal threading of the joint.

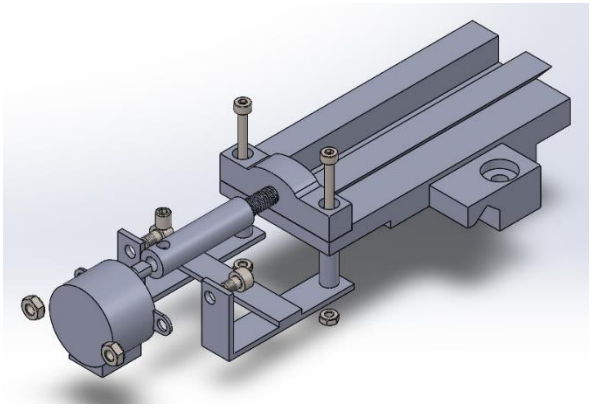


Figure 85: Stepper motor X-axis with clamps and joint

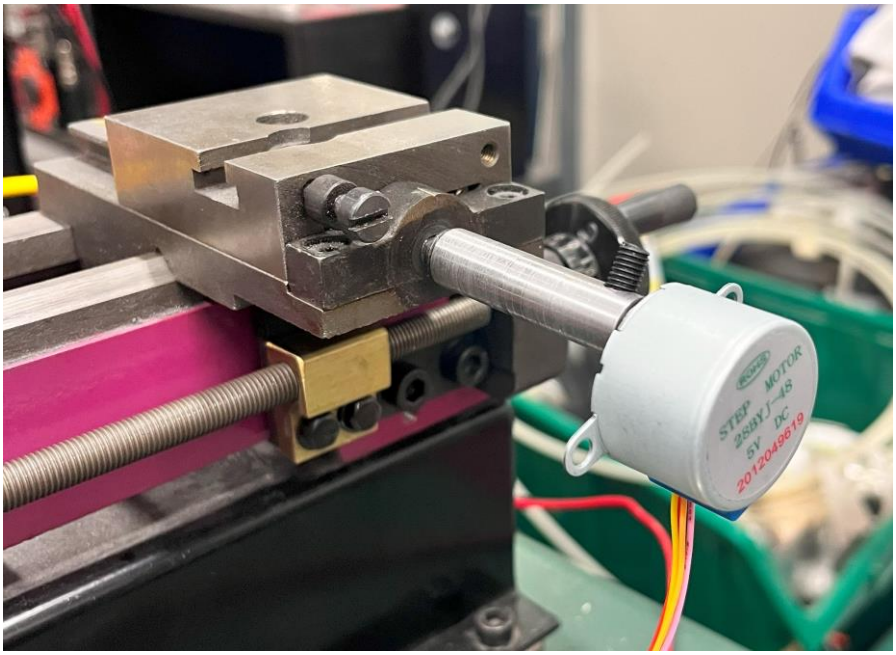


Figure 86: Mounting of the stepper motor on the carriage system

5. HMI platform (Fig. 87). The platform for the HMI features allocates the space for the membrane keypad (on the right side of the Fig. 87) and on the left side four cylindrical supports to place the main screen, fixed by 4 suitable ISO 7045 M3x20 screws and ISO 4032 M3 hex nuts.

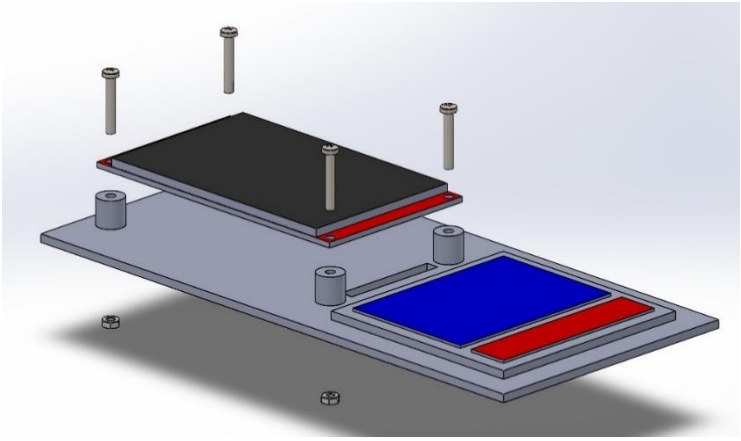


Figure 87: HMI platform

6. Ultrasonic sensor (Fig. 88). The final assembly includes the ultrasonic sensor, which is mounted on a suitable platform with 4 ISO 7045 M1,6x16 screws and ISO 4032 M1,6 hex nuts. The support is placed on the lathe bed, exploiting the presence of a threaded hole on the extreme side of the machine (ISO 4762 M5x12 screw).

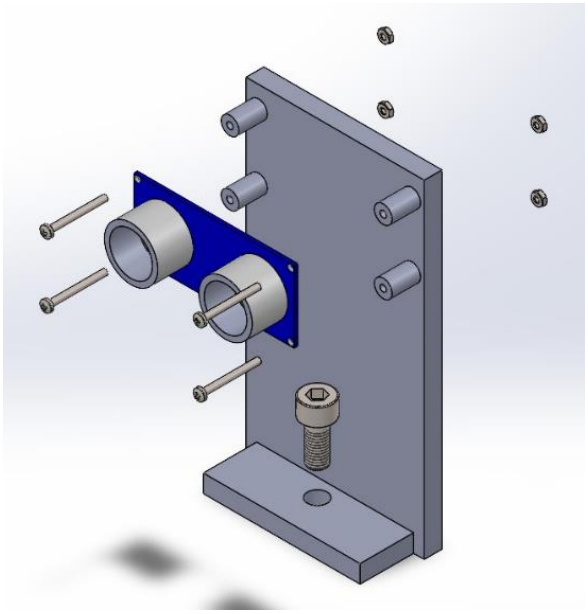


Figure 88: Ultrasonic sensor with the support

An important aspect to take into account for the sensor is the offset between the initial position of the carriage and the position of the sensor: this offset value must be included inside the Arduino code for the measurement of the positioning distance for the tool.

6. Conclusions and further improvements

This thesis project aimed to realize a preliminary automation of the work on a lathe machine, with the usage of the Arduino controller and the creation of a series of simple work programs that can be performed by the machine.

Many improvements can be made in order to create a more automated and versatile machine.

The first one is the design and realization of an electrical network for the DC motor able to handle a greater power and so to produce a higher rotational speed for the workpiece. Possible ideas can be a boost converter, such as the one presented above, but designed in a more appropriate way choosing the right components in order to operate in a safe environment. Another option can be to exploit the alternate current (230 V AC at 50 HZ of the electrical line) as direct power source for the supply circuit.

The network implemented in this project had the goal to use the Pulse Width Modulation technique with the Arduino and, due to its limited speed range, to test the processes developed at code level.

Furthermore, a second area of improvements regards the list of the mechanical turnings available on the machine, such as the parting and groove turning, flat facing and profiling with form tools, able to handle custom and more articulated shapes; adding a tailstock, the number of possible processes increases because the internal operations will be available in addition with the work on long work pieces. Accordingly, the external and internal threading and the drilling turnings are viable. At control level, this new component requires a dedicated control even if it remains manually or automatically actuated.

General improvements can be carried to all the stages of the codes and at component choice level, such as regarding to the stepper motors specifications and sensors applied, to obtain a larger variety for the materials to be machined.

Finally, as the modern CNC machines, the last functionality that can be developed in order to level up the performances and results of the machine is the development of an adaptive control, especially for the spindle velocity. As analysed in the related chapter before, the rotational speed depends inversely on the work diameter and directly on the cutting speed, which is assumed constant during the whole work cycle. From the relation it can be deduced that, as the diameter of the piece being machined varies, the rotation speed of the piece must vary, automatically controlled by the CNC as it receives the signals emitted by the linear scale reader Ax, integral with the tool holder slide that transmits each variation in diameter of the piece. The constant cutting speed of the tool, achieved only with the advent of numerical control, represents the ideal machining condition that allows obtaining machined surfaces with an excellent degree of finish (almost like grinding) and a longer life of the cutting capacity of the tool.

Bibliography

[01] Ye, R. (2024, Jan 26). Retrieved from https://www.3erp.com/blog/lathe/

[02] C. Di Gennaro, A. L. Chiappetta, A. Chillemi. (2012). *Nuovo corso di tecnologia meccanica, Metallurgia delle polveri - Diagrammi di equilibrio - Trattamenti termici - Lavorazioni per asportazione di truciolo.* Hoepli.

[03] C. Di Gennaro, A. L. Chiappetta, A. Chillemi. (2012). *Nuovo corso di tecnologia meccanica, Qualità e innovazione dei prodotti e dei processi.* Hoepli.

[04]. (n.d.). *Arduino datasheet.* Retrieved from https://docs.arduino.cc/resources/datasheets/A000067-datasheet.pdf

[05]. (n.d.). Retrieved from PWM technique: https://docs.arduino.cc/tutorials/generic/secrets-of-arduino-pwm/

[06] Alberti, D. (n.d.). Retrieved from https://www.danielealberti.it/2017/02/modificare-frequenza-del-pwm-di-arduino.html

[07] Alfieri, M. (n.d.). Retrieved from https://www.mauroalfieri.it/elettronica/frequenza-pwm-arduino-duty-cycle.html

[08]. (n.d.). Retrieved from UART serial protocol: https://docs.arduino.cc/learn/communication/uart/

[09]. (n.d.). Retrieved from I2C serial protocol: https://docs.arduino.cc/learn/communication/wire/

[10]. (n.d.). *LCD Wiki.* Retrieved from TFT touch screen display: http://www.lcdwiki.com/res/MSP4021/4.0inch_SPI_Module_MSP4020&MSP4021_User_Manual_EN.pdf

[11]. (n.d.). Retrieved from Liquid Crystal Display guide: https://docs.arduino.cc/libraries/liquidcrystal/

[12]. (n.d.). Retrieved from Membrane Keyboard guide: https://circuitdigest.com/microcontroller-projects/interface-4x4-membrane-keypad-with-arduino

[13]. (n.d.). Retrieved from How to Mechatronics - Ultrasonic sensor guide: https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/

[14]. (n.d.). Retrieved from How to Mechatronics - Stepper motors guide: https://howtomechatronics.com/tutorials/arduino/stepper-motors-and-arduino-the-ultimate-guide/

[15] Cavagnino, A. (2005/2006). *Electrical machines.*

[16] Storey, N. (2013). *Electronics: A system approach* (5th ed.). Pearson Education Limited.

[17]. (n.d.). *Curio Res*. Retrieved from Encoder guide (1): https://www.youtube.com/watch?v=dTGITLnYAY0&list=PLoslJMbwnVd-_6rGpDPYtKONACo7sUntX

[18]. (n.d.). *Curio Res*. Retrieved from Encoder guide (2): https://www.youtube.com/watch?v=HRaZLCBFVDE&list=PLoslJMbwnVd-_6rGpDPYtKONACo7sUntX&index=2

[19] Testolin, G. (n.d.). *Controllori PID e tecniche anti wind-up.*

[20]. (n.d.). Retrieved from Boost converter guide (1): https://www.monolithicpower.com/en/power-electronics/dc-dc-converters/boost-converters

[21]. (n.d.). Retrieved from Boost converter guide (2): https://journals.indexcopernicus.com/api/file/viewByFileId/258914

[22]. (n.d.). Retrieved from Boost converter guide (3): https://www.ti.com/lit/an/slva061/slva061.pdf?ts=1715007819226

[23]. (n.d.). Retrieved from Boost converter guide (4): https://www.ti.com/lit/an/slva372d/slva372d.pdf?ts=1714983227182&ref_url=https%253A%252F%252Fwww.google.com%252F