Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: informatica

Masterthesis

Development and implementation of a prototype pick-and-place machine

Simon Knuts

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: informatica

PROMOTOR :
Prof. dr. Kris AERTS

Gezamenlijke opleiding UHasselt en KU Leuven

Universiteit Hasselt | Campus Diepenbeek | Faculteit Industriële Ingenieurswetenschappen | Agoralaan Gebouw H - Gebouw B | BE 3590 Diepenbeek

Universiteit Hasselt | Campus Diepenbeek | Agoralaan Gebouw D | BE 3590 Diepenbeek
Universiteit Hasselt | Campus Hasselt | Martelarenlaan 42 | BE 3500 Hasselt

2023
2024

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: informatica

Masterthesis

Development and implementation of a prototype pick-and-place machine

Simon Knuts

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: informatica

PROMOTOR :
Prof. dr. Kris AERTS

UHASSELT KU LEUVEN

Preface

Throughout the last semester, I have learned a lot and had the pleasure of meeting many great people. Therefore, I would like to take this opportunity to acknowledge a few individuals and organizations for their contributions to this thesis and my personal growth.

First, I would like to extend my appreciation to the UPV, KU Leuven and UHasselt for giving me the opportunity to live and study a semester in Valencia.

I would like to express my gratitude towards my supervisor Juan José Serrano Martín, both for his guidance throughout this project and his uplifting and encouraging spirit. To prof. dr. Kris Aerts for proofreading my master's thesis and his support throughout this project and my entire academic career.

I would also like to thank my roommates for lifting my spirits during times of struggle.

Last but not least, I want to thank my family at home. Thank you for all the opportunities you have given me in life and for being there to support me through it all. I am really lucky and grateful to have you.

Contents

Acronyms

AC Alternating current
CAD Computer-aided design
CAM Computer-aided manufacturing
CLK Clock
CNC Computer numerical control
CS Chip select
DC Direct current
EMI Electromagnetic interference
GPIO General purpose input/output
HAL Hardware abstraction layer
I/O Input/output
IDE Integrated development environment
LL Low layer
MISO Master in slave out
MOSI Master out slave in
NEMA National electrical manufacturers association
PCB Printed circuit board
PWM Pulse width modulation
RAM Random-access memory
RPM Revolutions per minute
RX Receive data
SCK Serial clock
SDI Serial data in
SDO Serial data out
SPI Serial Peripheral Interface
SS Slave select
SYSCLK System clock
TTL Transistor-transistor logic
TX Transmit data
UART Universal asynchronous receiver/transmitter

List of Tables

List of Figures

Abstract

The growing importance of electronic devices in today's rapidly advancing society has led to an increasing demand for printed circuit boards. Consequently, the demand for faster and more accurate pick-and-place machines, which are utilized in printed circuit board assembly lines worldwide, is steadily rising. Although several industrial-grade options are available on the market, acquiring a machine that balances low cost with a reasonable accuracy remains challenging. This thesis investigates the feasibility of developing and implementing a prototype pick-and-place machine that fulfils these requirements. To accomplish this, the benefits and drawbacks of various mechanical and electronical components were first explored. Once viable components were identified, the machine was designed, assembled, and programmed. Subsequently, its performance was evaluated and compared with that of existing machines. This research and development process resulted in a machine capable of reliably placing the most commonly used components at a reasonable speed. In its current state, the machine can be constructed for less than 1000 euros, making it more affordable than most low-end alternatives currently available. Nonetheless, performance-wise, there is room for improvement. For instance, the speed and accuracy can be cost-effectively optimised by addressing bottlenecks. Although the machine presented in this thesis is still a prototype, it demonstrates considerable promise for further development.

Abstract in Spanish

La creciente importancia de los dispositivos electrónicos en la sociedad actual, y la velocidad con la que esta avanza, ha provocado un aumento en la demanda de placas de circuitos impresos. En consecuencia se ha generado una demanda de máquinas pick-and-place más rápidas y precisas, que se utilizan en las líneas de montaje de placas de circuitos impresos de todo el mundo, esta demanda no deja de aumentar. Aunque en el mercado existen varias opciones de calidad industrial, la adquisición de una máquina que equilibre un bajo coste con una precisión razonable sigue siendo un reto. Esta tesis investiga la viabilidad para desarrollar e implementar un prototipo de máquina pick-and-place que cumpla estos requisitos. Para ello, se estudiaron las ventajas e inconvenientes de diversos componentes mecánicos y electrónicos. Una vez identificados los componentes viables, se diseñó, ensambló y programó la máquina. Posteriormente, se evaluó su rendimiento y se comparó con el de las máquinas existentes. Este proceso de investigación y desarrollo dio como resultado una máquina capaz de colocar de forma fiable los componentes más utilizados a una velocidad razonable. En su estado actual, la máquina puede construirse por menos de 1.000 euros, lo que la hace más asequible que la mayoría de las alternativas de gama baja disponibles en la actualidad. No obstante, el rendimiento de la máquina es mejorable. Por ejemplo, la velocidad y la precisión pueden optimizarse de forma rentable solucionando los cuellos de botella. Aunque la máquina que se presenta en esta tesis es por el momento un prototipo, promete mucho de cara a futuros desarrollos.

Abstract in Dutch

Het toenemende belang van elektronica in onze snel evoluerende samenleving heeft geleid tot een stijgende vraag naar printplaten. Bijgevolg is er een behoefte aan steeds snellere en nauwkeurigere pick-and-place machines, die wereldwijd gebruikt worden in assemblagelijnen voor printplaten. Hoewel er verschillende industriële opties op de markt zijn, blijft het een uitdaging om een machine te vinden die een lage kostprijs combineert met een acceptabele nauwkeurigheid. Deze thesis onderzoekt de haalbaarheid van het ontwerpen en implementeren van een prototype pick-and-place machine die aan deze eisen voldoet. Hiervoor werden eerst de voor- en nadelen van verschillende mechanische en elektronische componenten onderzocht. Na het identificeren van geschikte componenten werd de machine ontworpen, geassembleerd en geprogrammeerd. Vervolgens werden de prestaties geëvalueerd en vergeleken met reeds bestaande machines. Dit onderzoeks- en ontwikkelingsproces resulteerde in een machine die de meest gebruikte componenten betrouwbaar kan plaatsen aan een acceptabele snelheid. In zijn huidige staat kan de machine gebouwd worden voor minder dan 1000 euro, waardoor deze betaalbaarder is dan de meeste low-end alternatieven die momenteel verkrijgbaar zijn. Prestatiegewijs is er echter nog ruimte voor verbetering. Zo kunnen de snelheid en de nauwkeurigheid kosteneffectief worden geoptimaliseerd door knelpunten aan te pakken. Hoewel de gepresenteerde machine nog een prototype is, toont deze reeds potentieel voor verdere ontwikkeling.

Justification

This project serves as the culmination of my Master's studies in Software Systems Engineering Technology at the KU Leuven and UHasselt. It was conducted as part of an Erasmus exchange at the Universidad Politécnica de Valencia, where the project belongs to the field of Mechatronics Engineering. The subject of the project was proposed by prof. Juan José Serrano Martín and includes designing and implementing a prototype pick-and-place machine.

While the emphasis of this project lies in developing and assessing the feasibility of a cost-effective pick-and-place machine, the resulting system can also be used as a foundational resource for future mechatronics students. By acting as a base for further development, the system will enable students to increase their knowledge in this domain.

Chapter 1

Introduction

1.1 History of pick-and-place

In the beginning of the 20th century, electronic circuits were designed and constructed from small integrated circuits and discrete components. These components and their wired connections were then mounted to a rigid substrate and soldered by hand. The circuits were large, heavy, and relatively fragile. Manufacturing was slow and debugging was difficult [28]. As time went on, the demand for smaller and more complex electronic circuits rose. It became exponentially more difficult to manually mount and connect all the components of these circuits together with soldered wires. The first iterations of printed circuit boards (PCB) solved this problem and revolutionized the scene. On a printed circuit board, the copper connections, also called traces, are deposited directly on insulating substrates and components are mounted on connection points, also called pads [28, p. 682,691]. While the connections didn't need to be soldered manually any more, the placement of the components did. And as electronic devices continued to shrink, doing this manually became almost impossible. This is when the first pick-and-place machines were invented and used to accurately place these components on their dedicated pads on the PCBs. The first iterations of these machines were relatively basic, operating at a maximum placement speed of a few components per minute [29]. Since then, technology advanced significantly, and modern iterations can place up to 3000 components per minute with high accuracy [30]. Today, PCB's power everything from smartphones to traffic lights to airplanes. Standard designs can have thousands of small passive components, advanced integrated circuits with very high pin counts and numerous connections between these components. As demand for these complex printed circuit boards keeps steadily increasing, the demand for faster and more precise pick-and-place machines is increasing as well.

1.2 Working of a pick-and-place machine

Modern pick-and-place machines use a head piece equipped with one or more vacuum-controlled nozzles to pick up components from a supply reel or tray and accurately position them onto designated pads on the PCB. While industrial grade machine often times employ a head equipped with multiple nozzles to pick up various components, lower-end pick-and-place machines often utilize a single-nozzle head. To maintain the same functionality, this single nozzle can be automatically swapped using a nozzle changer mechanism. To ensure that each component adheres to its pad once placed, solder paste must be applied onto each of the pads beforehand. While some machines use solder paste depositing nozzles to accomplish this task, this tends to slow down the placement process. Consequently, separate machines which use stencils to deposit solder paste onto the pads are mostly utilized in the industry [31].

Incorporated within the pick-and-place machines is a machine vision system that captures images of the components after they are picked up. These images are used to rotate each nozzle, ensuring that every component is precisely aligned at the correct angle to match its corresponding pad on the board. Additionally, in case the image indicates that a component is defect, the machine will reject it and retrieve the next component from the feeder [31].

To facilitate these operations, the head should be able to move freely along at least three axes and rotate around the axis on which the head is located. Each of these axes is powered by motors which can be independently controlled by a computer numerical control (CNC) system, which guides the movement of the pick-and-place head, ensuring the precise placement of the components. The CNC system operates by interpreting machine control instructions, typically in the form of G-codes, and executing them sequentially. These machine control instructions are automatically generated by using a combination of computer-aided design (CAD) and computer-aided manufacturing (CAM) software. Starting with a PCB schematic as input, this software generates a tool path that dictates the sequence of movements the pick-and-place machine should follow to place all the parts. Subsequently, this tool path is translated into machine control instructions and delivered to the machine for execution [32]. An example of a modern pick-and-place machine can be seen in figure 1.1.

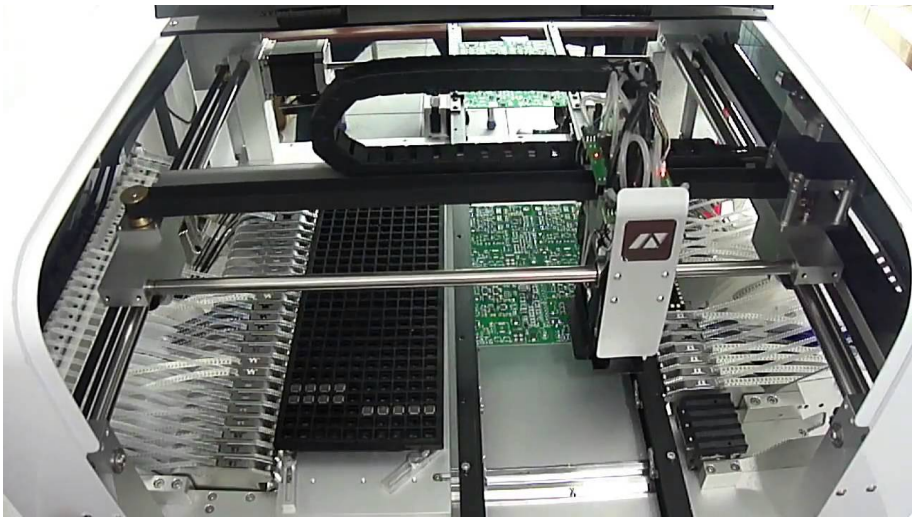

Figure 1.1: Modern-day pick-and-place machine [2]

Chapter 2

Objective

Today, PCBs have become basic components of our everyday life, and the appeal and capability of manufacturing them is no longer confined to large corporations with budgets for industrial grade machines. Small companies, hobbyists, and even individuals are increasingly searching for cost-effective ways to produce them in-house. Consequently, the demand for low-end pick-and-place machines that can accurately place complex components is steadily rising.

While several low-end, ready-to-use machines are currently available ([33], [34]), there remains significant potential for improvement and innovation in this regard. This thesis seeks to contribute to this field by developing and implementing a prototype pick-and-place machine utilizing an alternative design. The primary aim of this project is investigating the feasibility of this design by implementing it in practice and testing its viability.

The machine should be able to operate with an accuracy of 0.05-0.2 millimetres in order to place the most commonly used components [35]. This accuracy should be obtained with high precision while remaining cost-effective. The placement speed of the machine should be optimized but never negatively impact this accuracy.

The project will be executed in several stages. First, the alternative design will be introduced and its advantages and disadvantages will be discussed. Then, a component selection phase will commence, which includes researching and selecting components that meet the design requirements. Both the electronic and software systems architectures will be designed and subsequently constructed. Then, the final machine will be assembled using the selected components and constructed subsystems. Testing and evaluation will be performed to assess the machine's accuracy, precision, and overall performance through standard pick-and-place actions. Finally, the cost-effectiveness of the machine will be assessed by constructing a comprehensive financial breakdown of all parts used in the project.

Chapter 3

Initial design

Many low-end pick-and-place machines utilize a hierarchical structure where each axis is mounted on another. Specifically, the A-axis is mounted on the Z-axis, the Z-axis is mounted on the Y-axis, and the Y-axis is mounted on the X-axis. Examples of such machines include the Liteplacer [33] and the A01-Microsmt pnpV3 [34]. This design is standard and has the benefit that the placement speed is only dependent on the movement speed of the head. However, a downside of this design is that it can become costly and complex. The main reason for this is that the motor controlling the X-axis movement needs enough power to move the combined weight of the Y-, Z-, and A-axes.

The prototype presented in this thesis aims to cut costs by employing an alternative design. It uses a head that moves in a 2D plane combined with a moving floor in order to reach all positions. In this setup, which can be seen in figure 3.1 the X-axis stepper motor only needs to move the weight of the Z- and A-axes. A potential downside of this approach is that moving the floor, which holds the PCBs, might cause them to shake. This is unacceptable in any pick-and-place application, as it significantly impacts the precision and accuracy of the machine. However, since we are working with a prototype machine, the placement speeds will not be very fast. Therefore, in case this issue arises, we expect to mitigate it by making the floor slightly adhesive.

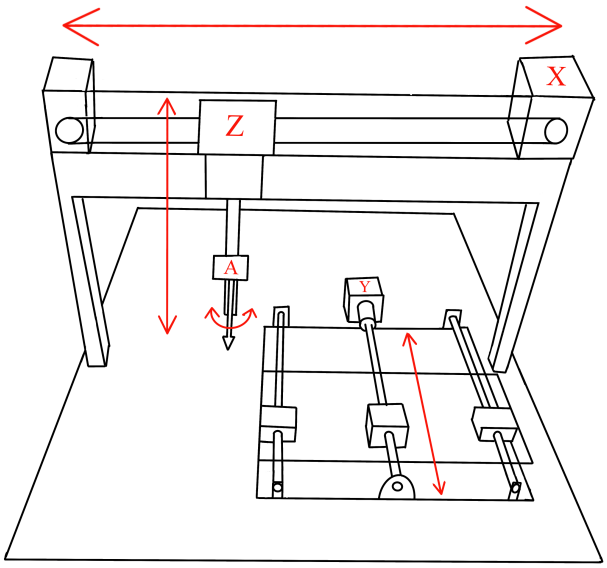

Figure 3.1: Initial design of the pick-and-place machine

Chapter 4

Selection of components

Every building process begins with the careful selection of the right materials. For this machine, these components need to ensure that the machine remains cost-effective while also being stable, compact, fast, and accurate. The components used in this machine are a blend of repurposed ones, no longer in use and newly ordered components, ensuring sustainability and cost efficiency in this early exploration phase.

4.1 Mechanical components

While all mechanical components will be considered in the financial analysis of the project, only the most important components will be discussed in this chapter, as many of them are fairly standard.

4.1.1 T-profiles

T-slot aluminium construction profiles will provide the mechanical base of the pick-and-place machine. These profiles are lightweight yet offer a high level of rigidity, which is a crucial aspect for CNC machines. Rigidity ensures that the supporting frame remains stable during operation, preventing any shaking or vibrating that could compromise accuracy and precision.

Figure 4.1: T-slot aluminium profile [3]

4.1.2 Ball screws

A ball screw is a mechanical linear actuator that translates rotational movement into linear movement. It operates by running ball bearings along a helical threaded shaft, so they act as a precision screw [36].

While their working principle is similar to that of regular lead screws, they have significantly lower friction. This is achieved by using ball bearings instead of letting the nut and screw shaft move directly against each other. This low level of internal friction allows ball screws to achieve a very high level of efficiency and positional accuracy even at high torque and force loadings [37].

Thanks to its high level of positional accuracy, this mechanical actuator will be used to move the loads on both the Y- and Z- axis of the machine.

Figure 4.2: Ball screw actuator [4]

4.1.3 Belt drive

A belt drive also is a mechanical linear actuator that translates rotational movement into linear movement. It operates by using a belt connected between two circular pulleys and rotating one of those pulleys to move the belt [38]. To efficiently transfer torque between the pulleys, the belt contains teeth that fit into grooves of the pulleys. These teeth also provide the necessary friction to prevent the belt from slipping. An important benefit of belt drives over ball screws is that they are generally more cost-effective. Especially when the goal is to move loads over longer distances, this discrepancy becomes substantial [39].

While ball screws can generally provide a higher degree of accuracy, using a belt drive for the X-axis is much more cost-effective. Additionally, the level of accuracy achieved by a belt drive is expected to be sufficient for the application. Therefore, a belt drive will be utilized to move loads along the X-axis.

Figure 4.3: Belt drive actuator [5]

4.1.4 Guide rails

The head of the pick-and-place machine will contain a significant amount of weight. Round bars that act as linear guide rails will help support this weight, so the machine stays stable during movements and accuracy doesn't decrease.

Figure 4.4: Guide rails [6]

4.1.5 Cable carrier

A cable carrier is a chain of links in which the cables of the head piece can be put, so there is no possibility of them getting stuck on various parts of the machine when the head is in motion.

Figure 4.5: Cable carrier [7]

4.1.6 Nozzles

The nozzle is a crucial part in pick-and-place machines, as it is utilized to pick up and release the components. There are many variations of nozzles available, each designed to accurately and precisely place components of specific shapes and sizes. For this project, several variations of Juki nozzles will be used, as they offer high quality at a low price point.

Figure 4.6: Juki pick-and-place nozzles [8]

4.1.7 Nozzle holder

A nozzle holder allows a pick-and-place machine to swap nozzles without manual assistance from the operator. It holds the nozzles in a specific manner that allows the machine to swap them using a relatively small number of motor motions. The holder has a specific, established place on the machine in order to streamline this swapping process. This holder did not have to be designed from scratch, as multiple designs are available online. For this project, a 3D-pritable design, sourced from [9], was used. The selected design can be seen in figure 4.7 and is capable of holding three Juki nozzles.

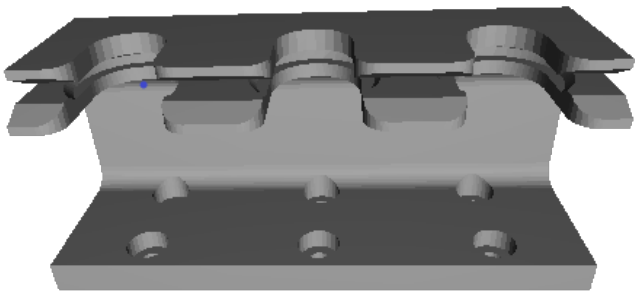

Figure 4.7: Juki pick-and-place nozzle holder [9]

4.1.8 Shielded cables

The final machine will include numerous cables connecting various components. Consequently, electromagnetic interference (EMI) between these cables could cause disruptions in the operation. In order to prevent this, the use of shielded cables is necessary, as they insulate the cable and reduce the effects of the EMI [40].

4.2 Electromechanical components

4.2.1 Selection of the type of motor

The head of the machine must be capable of moving along three axes (X, Y and Z) and rotating along the Z axis. This movement will be driven by four motors, each specifically selected to fit the requirements of its corresponding axis. When selecting the correct motor for any motion control application, a few important parameters to consider include:

- Speed : "The speed of a motor is defined as the rate at which the motor rotates. The speed of an electric motor is measured in revolutions per minute (RPM)." [41]
- Torque: "The torque output of a motor is the amount of rotational force that the motor develops." [41] The torque of a small electric motor is commonly measured in Newton centimetres (N.cm).
- Accuracy: "the ability of a motor to achieve an exact rotational position." [42]
- Precision: "the ability of a motor to consistently repeat the desired motion." [42]
- Cost: the cost of the motor.

Motor type comparison

Before comparing the aforementioned factors of different motor models against the requirements, it is crucial that the right motor type for our specific application is selected first. For this decision, two types were considered: direct current (DC) motors and stepper motors.

The main difference between stepper motors and DC motors is their operation mode. Stepper motors are open-loop systems that move in discrete steps, with each step corresponding to a fixed angular displacement. In an open-loop system, the output has no influence on the input signal, as can be seen in figure 4.8. This means that the system cannot self-correct any errors it could make when its positional value drifts. While this is an important downside to consider, an advantage of the stepper motor is that it does not require feedback devices since it moves in discrete steps. As DC motors rotate continuously, they do need feedback devices such as an encoder to accurately calculate their position. Therefore, DC motors operate as a closed-loop system.

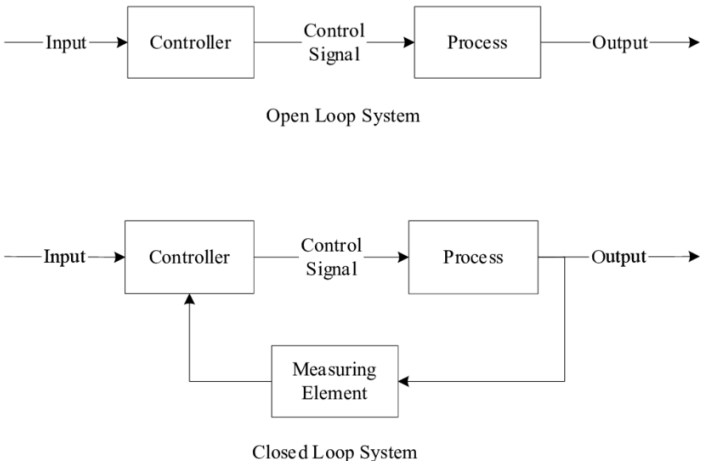

Figure 4.8: Comparison: open loop system vs closed loop system [10]

The amount of torque stepper motors are able to produce is inversely proportional with their speed [43]. Therefore, their maximum speed is highly dependent on the load, and for most applications will be limited to the range of 600 to 1500 RPM [44]. While the speed of DC motors also decreases as the load rises, they are more capable of producing a stable torque from a low speed range to high speed range. Depending on the operating voltage of the motor and its load, DC motors can reach maximum speeds in the range of thousands to ten thousands RPM [45].

Stepper motors are generally less efficient than DC motors, since they lose more energy through heat dissipation. Furthermore, stepper motors constantly draw their maximum supported current when operating, reducing their energy efficiency significantly. In DC motors, the current draw is based on the load and the efficiency of these motors has been highly optimized due to their maturity. These factors render DC motors the more efficient option in terms of both power consumption and power relative to cost [46].

Motor type selection

While stepper motors are not as efficient or as fast as DC motors, their high accuracy and precision without the need for feedback devices and high torque at low speeds makes them more fitting for cost-efficient pick-and-place applications.

An argument can be made that servo motors should also be considered for this application, as they offer a similar or slightly better performance than stepper motors at a comparable price point. The reason this type of motor was not included in the comparison was solely based on availability. Both stepper and DC motors were readily available in the laboratory, so it would be financially suboptimal to order a completely new set of motors for the potential of merely a marginal increase in performance.

4.2.2 The stepper motor

Working of a stepper motor

All electric motors, including stepper motors, have a stationary part, called the stator and a moving part, called the rotor. On the stator, which is located around the centrally located rotor, multiple toothed electromagnets are arranged [47].

By powering one or more of these electromagnets, a magnetic field is generated by the current flowing in the electromagnets coil and the rotor will align itself with this field. By sequentially powering different electromagnets in different phases, the rotor can thus be continuously rotated over small distances [48]. Each of those discrete, slight rotations is called a “step”, hence the name stepper motor.

Characteristics of a stepper motor

Aside from the aforementioned general parameters to consider when choosing a motor, stepper motor's unique structure introduces additional factors that also must be considered

- Step angle: "The angular rotation during one full step, generally given in degrees." [49]
- Holding torque: "The amount of torque needed in order to move the motor one full step when the windings are energized, but the rotor is stationary." [50]
- Detent torque: "The amount of torque the motor produces when the windings are not energized." [50]
- Friction torque: The amount of torque created due to friction created between the bearings [49].

Types of stepper motors

The characteristics mentioned in the previous section are substantially influenced by the specific implementation details of the motor. These different implementations mainly differ in the structure and configuration of the rotor and stator components.

The three main types of rotor are:

- Permanent magnet rotor: The rotor is a permanent ring magnet that is magnetized with alternating north and south poles positioned in lines parallel to the rotor shaft. This solution guarantees a good torque and a detent torque, with the drawbacks of having a lower speed and a lower resolution than the other types [51][52].
- Variable reluctance rotor: The rotor is constructed out of a multi-toothed iron core. While this implementation reaches a higher speed and resolution than the others, the amount of torque it can develop is often lower. Additionally, since there is no permanent magnet in this implementation, it also has no detent torque [52].
- Hybrid rotor: As its name suggest, this type of rotor is a hybrid between a permanent magnet and variable reluctance versions. The rotor combines the magnet from the permanent magnet and the teeth from the variable reluctance motors [51]. Structurally, it consists of two discs with alternating teeth that are magnetized axially [48]. This complex configuration makes it more expensive than the other types but allows for a higher resolution, speed, and torque. The higher performance in comparison to the other types makes this type the most popular.

Aside from these rotor configurations, the characteristics of the stator can also vary between different implementations. The main characteristics of the stator circuit include its number of phases and pole pairs, as well as its wire configuration. The number of phases corresponds to the number of independent coils on the stator. The number of pole pairs indicates how main pairs of stator teeth are occupied by each phase.

Since hybrid stepper motors have been successfully employed in similar projects by students at UPV, this type will be utilized in this project as well. For a hybrid stepper motor, the step angle can be calculated by using the following equation:

$Step\ angle = 360/(2 * number\ of\ phases * number\ of\ rotor\ teeth)$ [53]

The most common configuration for hybrid stepper motors features a rotor with 50 teeth and a bipolar stator. A bipolar stator has a single winding per phase, resulting in two phases and four pole pairs. Inserting these values in the aforementioned formula yields a basic step angle of 1.8 degrees.

This step angle can be confirmed systematically: energizing the stator windings causes the rotor to move one fourth of a tooth pitch to align with the energized poles. Given that the rotor has 50 teeth and moves 1/4 tooth pitch at a time, the motor needs 200 steps to complete one full rotation, resulting in a step angle of 1.8 degrees [52].

Selection of stepper motor

Many different sizes of hybrid stepper motors are available online. The National Electrical Manufacturers Association (NEMA) [54] has created a standardized size classification system based on specific criteria. Although this standardization primarily focuses on motor frame sizes, the size of a motor gives a strong indication of its power.

The X-axis of the pick-and-place machine will carry a significant load as the entire Z-axis is mounted upon it. Therefore, a stepper motor with sufficient power must be selected to control the movement along this axis. The detent and holding torque requirements for this motor are minimal, since it primarily controls horizontal movement, where the load during standstill is nearly negligible. The motor selected for this task is the JK57HS76-2804-05 bipolar stepper motor [55]. Both the Y- and Z-axis have to drive a significantly smaller load than the X-axis. Therefore, a medium size stepper motor can be used to control the movement along these axes. For the Y-axis, the 17HS8401—42HS48 bipolar stepper motor [56] is selected. For the Z-axis of the machine, the 42BYGH48 bipolar stepper motor [57] is selected. Along this axis, it is important that the motor can deliver enough detent and holding torque to hold the load in its place under the effects of gravity. Since the only functionality of the A-axis is rotating the attached components, a small size stepper motor can be utilized. The motor selected for this task was the SCA2818L1504-L bipolar stepper motor [58]. Table 4.1 displays the characteristics of the selected motors for all axes.

Table 4.1: Selected motors and their specifications for each axis

Controlled axis	X-axis	Y-axis	Z-axis	A-axis
Selected motor model	[55]	[56]	[57]	[58]
NEMA standard	NEMA 23	NEMA 17	NEMA 17	NEMA 11
Resolution (steps/revolution)	200	200	200	200
Current consumption per coil (A)	2.8	1.7	1.7	1.5
Rated voltage (V)	3	3	2.8	2.8
Winding inductance (mH)	3.6	3.2	2.8	1.9
Holding torque (N.cm)	189	52	55	18
Detent torque (N.cm)	5.9	2.6	2.6	0.8
Shaft diameter (mm)	6.35	5	5	8
Weight (g)	1100	350	300	200
Dimensions (mm)	56 x 56 x 76	42 x 42 x 48	42 x 42 x 48	28 x 28 x 52

4.2.3 Vacuum pump and solenoid valve

Both a vacuum pump and solenoid valve are crucial components for any pick-and-place machine, as they are needed to pick up parts. While there are many kinds of vacuum pumps available on the market, their working principle is all roughly the same. They work by converting energy into pressure by generating a partial or low-pressure vacuum by pushing gas or air molecules out of a sealed chamber [59]. This vacuum can then be transferred to the suction nozzles throughout a series of hoses and used to pick up a component. An intuitive approach to put down components would be to just simply turn the vacuum generator off. While this approach works, it is suboptimal as achieving a vacuum does not happen instantaneously. This means that by solely relying on the vacuum pump, a lot of time would be wasted waiting for a vacuum to form. This problem can be solved by placing a solenoid valve in between the vacuum generator and suction nozzles. The function of these valves is controlling the vacuum level and ensuring that it is transferred to the suction nozzles at the appropriate times.

For the vacuum pump, a DC 12V vacuum pump was used [11]. It has an inflation time of less than 10 seconds and is able to create a vacuum pressure of -58kpa. This pressure should be more than enough to pick up the small parts our machine will be working with [60].

Figure 4.9: DC 12V Vacuum pump [11]

The solenoid valve used in this project is a DC 12V, 2 way, zero differential solenoid valve which can be operated at 0 PSI (Vacuum) [12]. Since it is a zero differential solenoid valve, it does not need a differential pressure drop across the valve to work.

Figure 4.10: DC 12V Solenoid valve [12]

4.3 Electronical components

4.3.1 Stepper motor drivers

A stepper motor driver is essential for controlling a stepper motor. It works by converting the step and directions command pulses coming from a microcontroller into two sequenced phases and controlling the current of these phases [61]. To do this, it employs 2 H-bridge circuits, each connected to a different coil of the motor. Each H-bridge consists of 4 FET transistors with very low resistance between drain and source contact when in an active state [13]. By alternately closing the two pairs of transistors in the H-bridge circuit, we can create a current in the coils of the stators electromagnets that rhythmically changes polarity. When this process is happening in both H-bridge circuits in different phases, the step command pulse is successfully transformed into two sequenced phases. This process of rhythmically changing the current direction and its effects on the motor movement are visualized in figure 4.11 and 4.12 respectively.

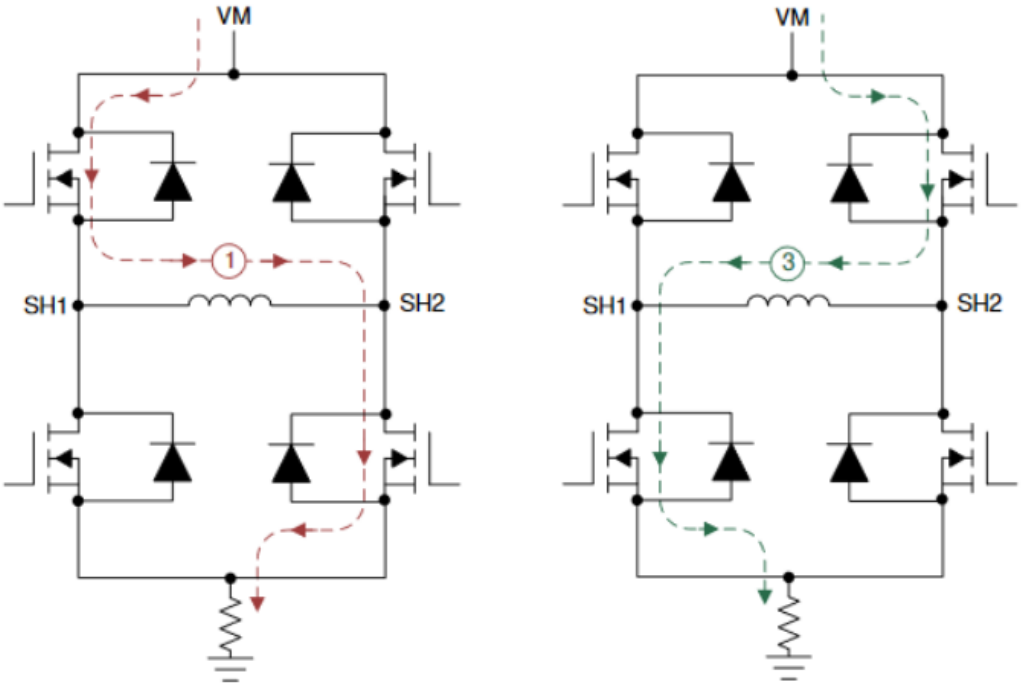

Figure 4.11: H-bridge circuits [13]

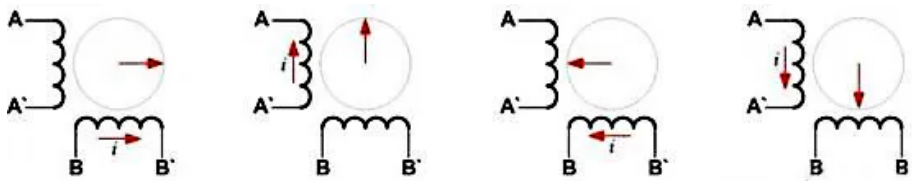

Figure 4.12: Movements of a stepper motor under different coil currents [14]

Microstepping

Most stepper motors have a step angle of 1.8 degrees, This is equivalent to 200 steps per revolution. While this resolution is adequate for most applications, it can be insufficient when very precise movement is required. In these applications, microstepping offers a solution. Microstepping enables a stepper motor to make more than 200 steps per revolution and in turn reduce its step angle.

When driving stepper motors with full steps, the output of the stepper driver looks like a square signal. This square shaped signal, which is displayed in figure 4.13, can result in rough movements and noisy motors [13].

Microstepping aims to drive motors with a current waveform that's sinusoidal. This means that instead of powering the motor coils fully or not at all, they can also be powered with intermediate current levels. Doing this positions the motor in intermediate positions in between two subsequent full steps, and thus allows the motor to have a resolution of more than 200 steps per rotation. As the microstepping value increases, the output signal increasingly resembles a sine wave and the motor moves more smoothly, as can be seen in figure 4.13.

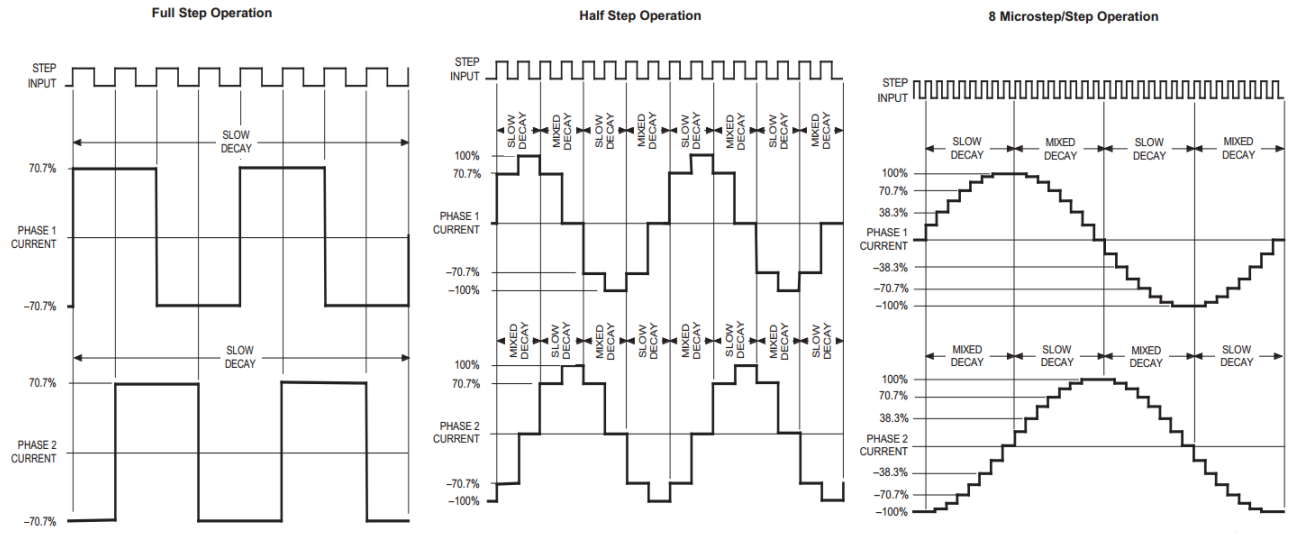

Figure 4.13: Shape of output signal in full-step, half-step and 1/8-step mode [13]

Microstepping also has an important downside, however. As mentioned before, each stepper motor is rated with a specific holding torque. This is also the torque required to pull the motor out of its current position. This holding torque decreases with microstepping because the motor is being held in place between full-step positions. Consequently, the magnetic paths will be longer and the holding torque will be reduced. [62].

As the holding torque in this scenario refers to the torque it takes to increment the position of the motor to the next full step, it is also called the incremental holding torque. This incremental holding torque can be calculated with the following equation:

$$Incremental\ holding\ torque = full\ step\ holding\ torque * sin * \frac{90°}{number\ of\ microsteps}$$ [63]

This means that, when performing a microstep, the torque load on the motor must be a fraction of the motor's rated holding torque. But even when there is no torque load on the motor after a step, a low incremental torque can still negatively affect the motor's accuracy. This is caused by

the fact that, even in standstill, a stepper motor still has a certain detent torque and a friction torque due to its bearings [63]. Therefore, if the mircrostep value is too great, it could happen that the motor can't produce enough incremental torque to overcome the bearing friction and detent torque and as a result not move.

Utilizing microstepping results in increased resolution and smoother motor movements. However, it does not directly increase the accuracy of a stepper motor, as the drop in incremental torque may allow for the axis position to be deflected. These advantages and disadvantages should and will be considered when deciding the optimal step modes for the motors.

Choice of stepper motor driver

The primary considerations to take into account when choosing the stepper motor driver are the voltage and current the driver can supply to the motor. It is important that these characteristics of the driver can at least match the current per phase and voltage requirements of the selected stepper motors. Depending on the application, the microstep options and motor control modes can also be an important factor to consider when selecting a stepper motor driver.

For this specific application there was access to three different stepper motor drives, namely: the DRV8825 [64], TMC2209 [24] and TB6600HG [65]. All three of these drivers are constant current drivers. This means that they apply a much higher voltage than the motor's rated voltage. This makes the current rise faster and decreases the influence from the generation voltage from the coil [66]. Therefore, constant current drives offer better torque and high-speed characteristics than constant voltage drives.

However, to not damage the motor while running it at higher voltages than it is rated for, the driver also needs to keep the current at a fixed level. This can be done using current limiting resistors, but these will dissipate a relatively high amount of heat and waste power. Chopper drivers offer a better way of limiting the current. These drivers get their name from the technique of rapidly turning the output power on and off (chopping) to control motor current. This allows the current to be limited without having to use resistors. Therefore, the total dissipated head and power consumptions will be lower on drivers that employ this technique. A downside of chopper drivers is that they are generally more expensive, since they require additional electronics to monitor the current and control the voltage switching [67].

While the DRV8825 driver uses current limiting resistors, both the TMC2209 and TB6600HG employ the "greener" alternative chopper technique.

The aforementioned voltage chopper also serves a secondary purpose in the TMC2209 driver. This driver offers a feature called "StealthChop" [24], whose main working principle is based on the voltage chopper included in the driver. This feature guarantees that the motor is absolutely quiet in standstill and in slow motion.

The TMC2209 even offers additional useful features like "StallGuard4" [24], which provides an accurate measurement of the load on the motor. This measurement can be used for stall detection as well as other uses at loads below those which stall the motor. One of these other uses is called "CoolStep" [24] and is a load-adaptive current regulator. It uses the stallGuard4 load measurements to adjust the current flowing to the motor in order to minimize the amount required in a specific load situation. This saves energy and keeps the components cool.

By researching the specifications of these three drivers using [64], [24] and [65], the following comparison can be made:

Table 4.2: Comparison of different stepper motor driver's specifications

Stepper driver comparison			
	DRV8825	TMC2209	TB6600HG
Logic voltage	2.5V-5.25V	3V-5.2V	3.5V-5V
Operating voltage	8.2V-45V	4.75-29V	9V-42V
Continuous current per phase	1.5A	2A	3.5A
Maximum current per phase	2.5A	2.8A	4A
Micro step options	Full, 1/2, 1/4, 1/8, 1/16, and 1/32	Full, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128 and 1/256	Full, 1/2, 1/4, 1/8, 1/16 and 1/32
Current adjustment	Clockwise add, counterclockwise decrease	Counterclockwise add, clockwise decrease	Counterclockwise add, clockwise decrease
Motor control modes	Dir/Step	Dir/Step, UART, Internal Step Pulse Generator	Dir/Step
Current formula	$I_{FS}(A) = \frac{V_{REF}(V)}{5 \times R_{SENSE}(\Omega)}$	$I_{FS}(A) = \frac{0.325(V)}{0.02(\Omega) \times R_{SENSE}(\Omega)} \times \frac{1}{\sqrt{2}} \times \frac{V_{REF}}{2.5(V)}$	$I_{FS}(A) = \frac{V_{REF}(V)}{3} \div R_{NF}$

For the motors controlling the movement along the Y- and Z-axis and the one controlling the rotation around the Z-axis, the TMC2209 arises as the best choice. This driver can deliver the maximum of 1.7A and 3V necessary to control these motors without the need for external cooling, in contrast to the DRV8825. Additionally, the extra microstep options, motor control modes and features make it the better choice out of the two drivers. This driver is displayed in figure 4.14.

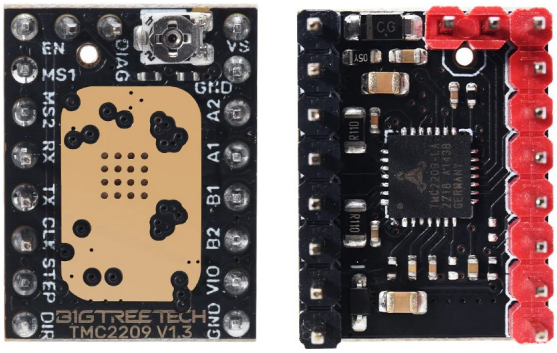

Figure 4.14: TMC2209 stepper motor driver [15]

Since the motor controlling the movement along the X-axis has a current per phase requirement of 2.8A, the more expensive TB6600HG driver will be selected for this purpose. This driver is displayed in figure 4.15.

Figure 4.15: TB6600HG stepper motor driver [16]

The TB6600HG can be completely configured using physical switches on its exterior. The TMC2209, however, can only partially be configured in this way by placing jumpers over specific pins. In order to have full control over the microstepping options and the other additional features mentioned, the TMC2209 requires a configuration through a universal asynchronous receiver/transmitter (UART) connection. This connection will later be explained in more detail.

4.3.2 Microcontroller

"A microcontroller is a compact, integrated circuit designed to govern a specific operation in an embedded system." [68] A microcontroller is often seen as a small computer, since it contains a processor, random-access memory (RAM), a flash memory, a serial bus interface and input/output (I/O) peripherals all on a single chip.

Explaining the functioning of a microcontroller in depth would deliver no added value to this thesis and lead us too far away from the essence of the project. Simplified, it works by storing both data and instructions it receives from its I/O peripherals in its data memory. This data memory can then be accessed by the processor to interpret the data and apply the instructions on them. The result of this process and the appropriate actions that have to be taken are then communicated to the necessary components by using the I/O peripherals once more [68].

In this project, the microcontroller will serve as the brain of the system, both controlling the electrical components and processing the feedback that it receives from them.

Choice of microcontroller

The microcontroller that was used for this project is the STM32F446RE mounted on its ARM STM32 Nucleo-64 development board [1]. Selecting this component was simple since the computational needs of the project are relatively limited and most modern 32-bit microcontrollers based on the Arm Cortex-M processor offer similar performance at a similar price point. Furthermore, this model has been used by students in previous, similar projects without any issues. This specific models offers the following specifications:

- 180 MHz max CPU frequency
- VDD from 1.7 V to 3.6 V
- 512 KB Flash
- 128 KB SRAM System
- 4 KB SRAM Backup
- Arduino Uno Revision 3 connectivity

While most of these specifications are somewhat standard, having the Arduino Uno connectivity is not something all microcontrollers offer. This added functionality will be useful for connecting the CNC shield to the microcontroller, which will be explained in the following chapter.

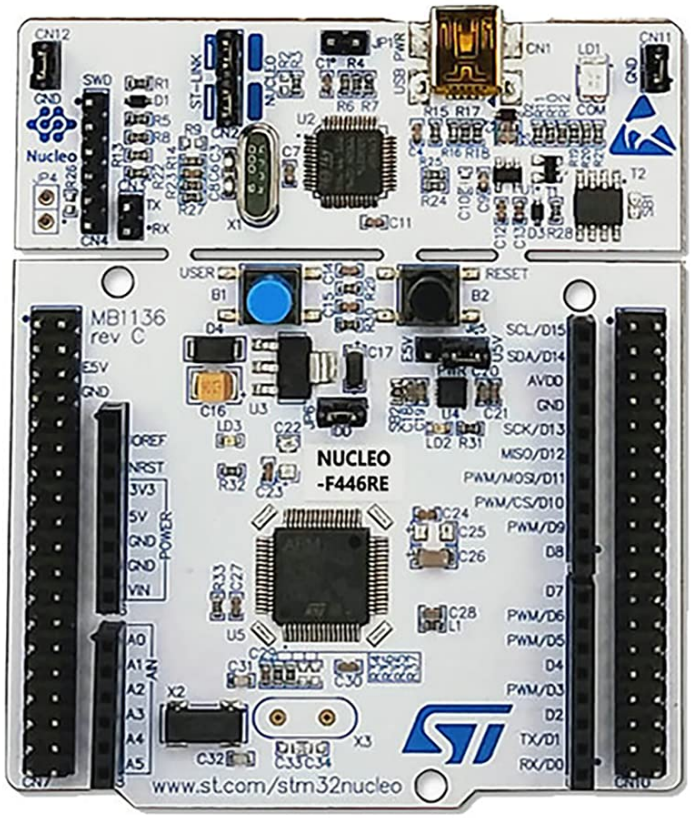

Figure 4.16: STM32 Nucleo-F446RE [1]

4.3.3 CNC shield

A CNC shield is a piece of hardware that can be added on top of an Arduino to control up to 4 stepper motors simultaneously. The specific CNC shield used in this project is the Protoneer CNC shield version 3.10 [69], which can be seen in figure 4.17. While this shield was originally designed to be used in combination with an Arduino microcontroller, it can also be used on our STM32F446RE due to its native Arduino cross connectivity.

The board was originally designed to use removable A4988 stepper controls, but it is compatible with a wide range of other stepper drivers. The shield consists out of 4 different driver boards corresponding to the X-, Y-, Z- and additional A-axes.

For each one of these axes, the shield contains both step and direction pins, which can be digitally controlled to move each one of the stepper motors in both directions. The shield contains pins which can be connected to buttons with different functions such as abort, stop, emergency stop, hold, and resume. The CNC shield also has pins for X-, Y-, and Z- end stops in both directions. The function of these end stops will be discussed in more detail in the next section. Finally, the CNC shield contains a separate 12-36 V power input specifically used to power all four of the stepper motors.

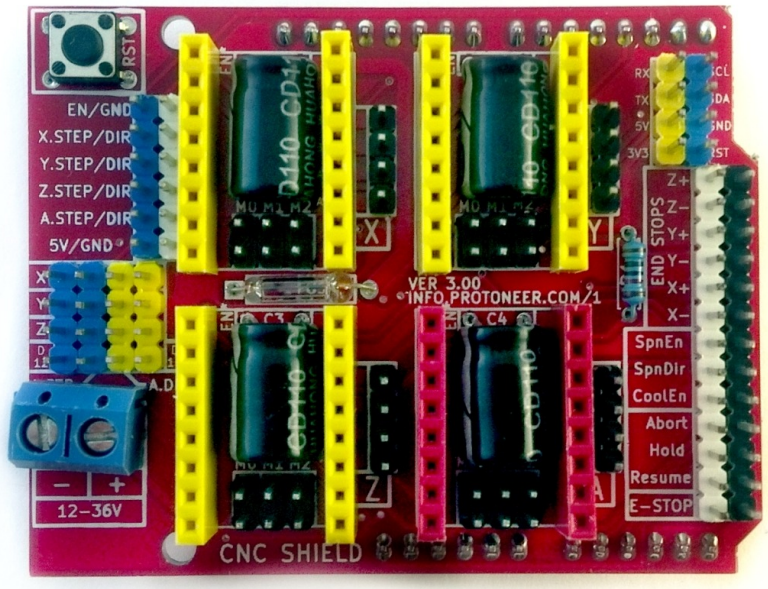

Figure 4.17: Protoneer CNC shield version 3.10 [17]

4.3.4 End stops

An end stop is a small and simple switch designed to be activated when the tool head hits one of its physical limits [70]. Generally, two of these are added to each one of the X-, Y- and Z-axes of the machine and their output signal is closely monitored by the microcontroller. In most implementations, the machine will automatically be stopped whenever this output signal is triggered. This functionality can prevent the machine from damaging itself when a failure of some sorts occurs and the machine moves past its limits.

These failures can have various causes such as missed steps, slippage, noise, mechanical failures and many more. Even in cases where these errors do not result in an end stop being triggered, they decrease the quality of the pick-and-place result. However, since the position of end stops is constant, they can function as a calibration tool to mitigate the effects of these errors and restore accuracy in the machine [71].

When operating in this context, the end stops are often called home switches and the process of calibrating the machine is called homing. This homing is mostly done during the startup process of the machine but can be done at any time throughout the operation. Because end stops and home switches share the same hardware, their digital signals upon activation are also indistinguishable. The only way these signals can be differentiated is by the predictability of them.

The homing of the machine will occur systematically at planned times throughout its operation. Therefore, the triggering of the end stops during this operation will be expected. Failures, on the other hand, can occur at random times and are often completely unpredictable. By setting a certain variable before the homing process, and disabling it afterwards, we can differentiate failures from homing. If an end stop is triggered and the variable is set, a homing sequence is happening, and the system can continue its operation. If an end stop is triggered and this variable is not set, a failure has occurred, and the system should stop the machine.

For this project, the mechanical end stops displayed in figure 4.18 will be used. These end stops are contact based and typically only connected with their signal and either the ground or input voltage. Whenever this type of switch is pressed, an electronical circuit closes and the signal is either pulled up or down depending on which pin is connected. While these components are cheap and work well for the most part, simple active contact switches will be susceptible to voltage spikes and EMI caused by other components [71]. This can cause the machine to act as if the switch has been unexpectedly pressed, thus imitating a signal from an end stop and ruining the result. This EMI can be minimized by careful wire routing, using shielded wires, ensuring proper grounding and employing filter circuits [40]. Most controller programs also contain a switch debounce routine that only validates a signal if it is held for a few milliseconds and ignores voltage spikes in this way.

Three wire non-contact end stops are generally more robust to voltage spikes and electrical interference since they actively pull the signal line high or low depending on the switch position. However, the added cost of using these switches does not result in noticeable benefits, since shielded wires and debounce routines will already be used.

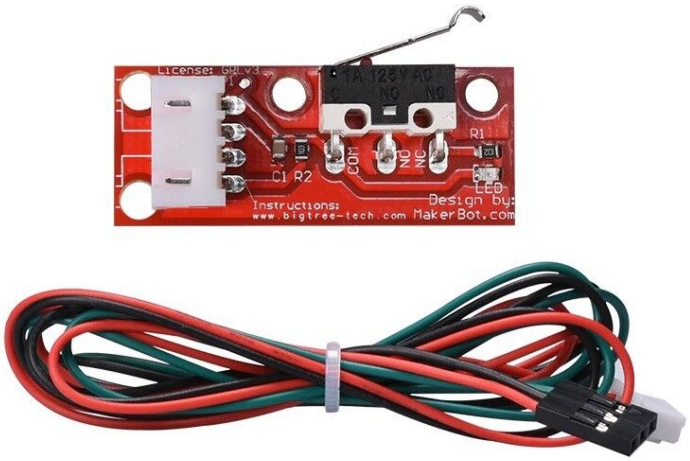

Figure 4.18: Mechanical end stop [18]

4.3.5 Display

Connecting a screen to our machine allows for information to be easily visualized. This includes feedback information, information indicating the current state of the machine, and even complex menus with multiple choices.

The specific model used in the project is the 1.8" 128×160 TFT LCD with SPI interface [72], displayed in figure 4.19. This model was selected because it was available in the laboratory and had been used in previous projects. This model uses 4-pin serial peripheral interface communication, has a 128 × 160 pixel resolution, and can display 18-bit of colours.

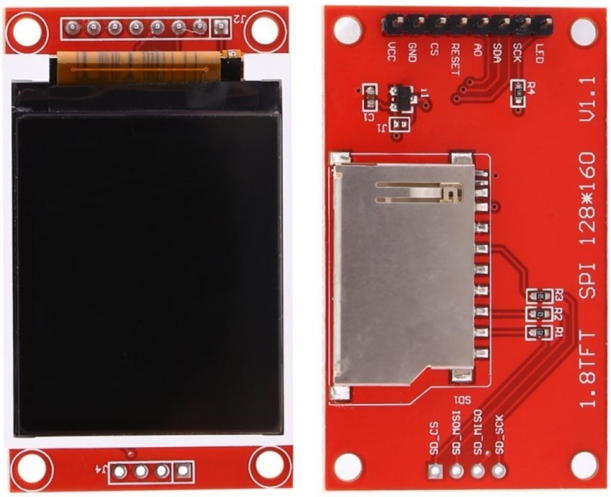

Figure 4.19: 128 × 160 TFT LCD with SPI interface [19]

4.3.6 Rotary encoder with switch

The system will require an input method since the screen will be displaying menus with multiple choices. This method needs to facilitate selecting different options as well as confirming this selection. Fortunately, these two functionalities are often combined in the same component in the form of a rotary encoder with a switch. The model selected for this project is displayed in figure 4.20.

A rotary encoder is a type of position sensor that converts the angular position of a knob into an output signal. This output signal can then be used to determine in which direction the knob is turned. Practically, this is possible because the encoder contains a disk with evenly spaced contact zones that are connected to one common and two separate pins. When the disk will start rotating step by step, the two separate pins will start making contact with the common pin and the two square wave output signals will be generated accordingly. The order of these generated pulses can be used to obtain the direction of rotation. Additionally, the rotated position can be obtained from this output by counting the pulses of the signal [73].

The functionality of a switch can be easily added to this encoder by making it such that when the knob is pushed down instead of rotated, a circuit closes and an electronic signal will be sent.

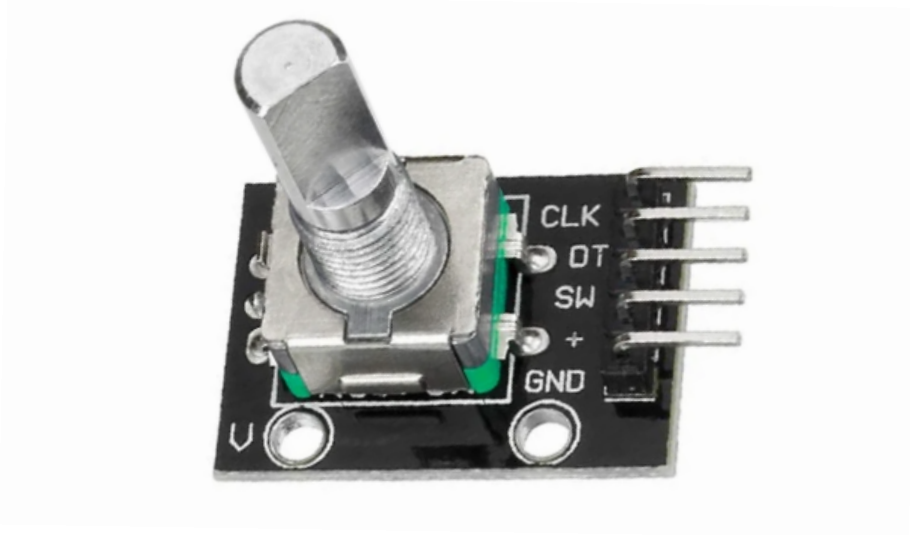

Figure 4.20: Rotary encoder with switch [20]

4.3.7 USB to TTL

A USB to TTL module is a serial adapter used to convert a USB interface to transistor-transistor logic (TTL) level. This module enables data communication between our STM32F446RE microcontroller, which only understand signals at TTL level, and a computer.

TTL is a low, non-differential voltage version of the RS-232 communication protocol [74]. In this protocol, a logical high is defined as either 5V or 3.3V and a logical low is 0V. While these lower voltages are useful for embedded applications, they make this specific protocol less robust to noise.

Because this module functions as a conversion tool between two different protocols, it is important that both devices use the same set of communication parameters. These parameters include:

- Baud rate: The rate at which information is transferred in a communication channel, commonly measured in bits per second [75].
- Number of data bits: The number of bits in each data frame or message, most binary protocols use 8 data bits [75].
- Number of stop bits: the number of stop bits used to mark the end of a frame, common values are one or two stop bits [75].
- Parity bit: The presence of this bit decides whether a data integrity check is included [75].

The specific converter used in this project is the FT232RL USB TO TTL 3.3V/5V FTDI Serial Adapter Module [21], which is displayed in figure 4.21 and is a standard and commonly used model.

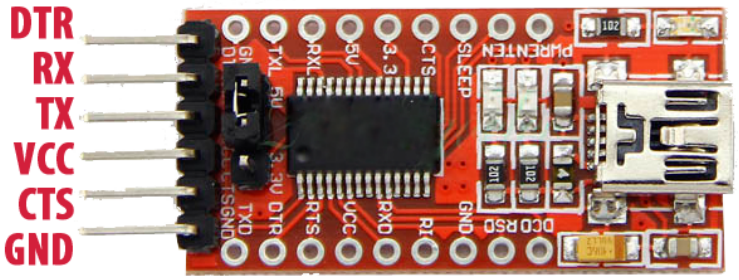

Figure 4.21: FT232RL USB TO TTL 3.3V/5V FTDI Serial Adapter Module [21]

This module is necessary since the machine codes to operate the machine will be sent from a computer to the microcontroller. This way, the machine can be operated in real time and G-codes don't need to be hardcoded in the microcontroller beforehand.

4.3.8 MOSFET voltage amplifier

Both the vacuum pump and solenoid valve will be digitally controlled by the microcontroller. However, since the microcontroller can only supply a maximum of 5V while both components operate at 12V, direct control is not possible. To address this, a voltage amplifier is used in the form of a switch MOSFET module [22], which can be seen in figure 4.22. These components are specifically designed to control high-power loads using low-power control signals. The module receives both a 12V DC power input from a power supply and a 5V control signal from the microcontroller. It then generates a 12V DC output signal that mirrors the control signal and is capable of operating both the vacuum pump and solenoid valve.

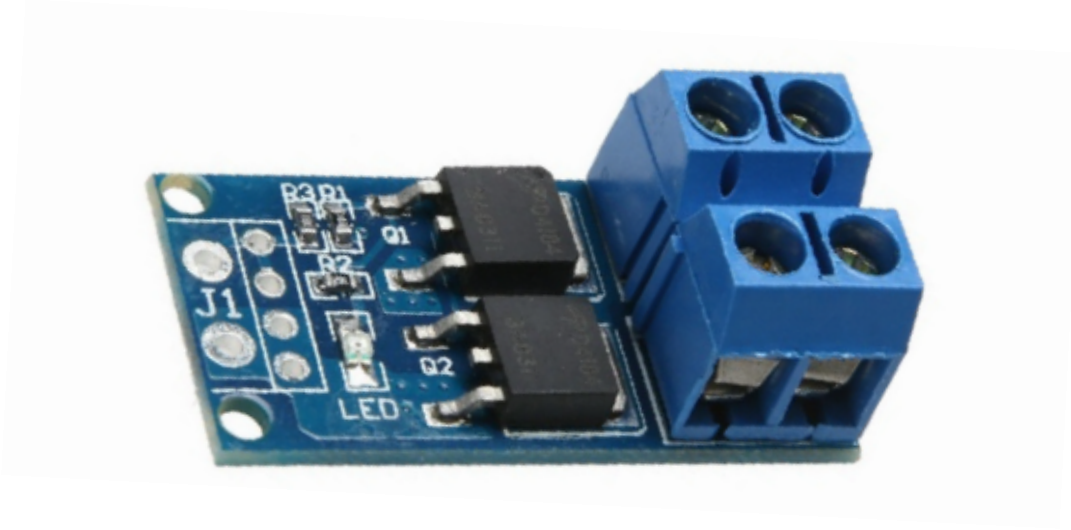

Figure 4.22: MOSFET voltage amplifier [22]

4.3.9 Power supply

A power supply is an electrical device that converts electric current from one source, mainly power plugs and sockets, to the voltage and current values necessary for powering a specific load [76]. Both the input and output of the power supply can be either alternating current (AC) or direct current (DC). In this project, the power supplies will be connected to main power or AC at 230V and a frequency of 50 Hz. They will be used to power various smaller electronic components, all working on direct current of different voltage levels. Therefore, the specific type of power supply necessary for this project is an AC/DC power supply. These power supplies can have either a linear or switching operating mode.

In power supplies with a linear operating mode, a transformer is used to reduce the AC input voltage to a level that is correct for the specific application. Then, the reduced AC voltage is rectified and transformed into a DC voltage. This DC voltage is then filtered a final time in order to further optimize the quality of the signal [76].

In switching AC power supplies, the voltage is rectified, filtered and transformed into a DC voltage at the input. Subsequently, it is transformed into a frequency pulse train by passing it through a chopper. As a last step, the pulse train is passed through another rectifier and filter in order to convert it back to a DC voltage. This final step is crucial, as it eliminates any remaining AC voltage component from the signal [76].

While linear power supplies definitely have some benefits, for example being relatively noise-free, they remain limited in terms of size and efficiency. This is caused by inherent design flaws, which make the miniaturization of their transformers practically impossible. In contrast, the design of switching power supplies allows the use of smaller transformers and therefore results in a smaller and lighter power supply. An additional advantage of switching power supplies is that they operate more efficiently than linear power supplies [76]. These factors make switching power supplies a more practical and efficient choice for many applications.

These reasons, combined with the availability of switching power supplies from previous projects, led to their selection for this project. The specific type of power supply used can be seen in figure 4.23.

In this project, a total of 3 power supplies will be used to power all the components:

- The CNC shield containing the three stepper motor drivers, which operates on 12-36V.
- The stepper motor driver responsible for the X-axis motor.
- Both the vacuum pump and solenoid valve operating on 12V.

While powering multiple of these components using a single power supply is possible, this approach is suboptimal. It can cause additional electromagnetic interference and noise between components that negatively impacts the overall functioning of the system [40]. Using three power supplies was recommended by the promotor of the project, as it strikes a balance between costs and avoiding interference.

Figure 4.23: Switching AC/DC power supply [23]

Chapter 5

System architecture

5.1 System schematic

The component selection phase was followed by the design and construction of the system architecture necessary for the project. An incremental approach was employed, where each component was sequentially connected and tested. This method ensured the functionality of the final system by initially testing all subsystems. However, several pin conflicts arose and had to be resolved during the process. Figures 5.1 and 5.2 illustrate the connections between all electrical and electromechanical components.

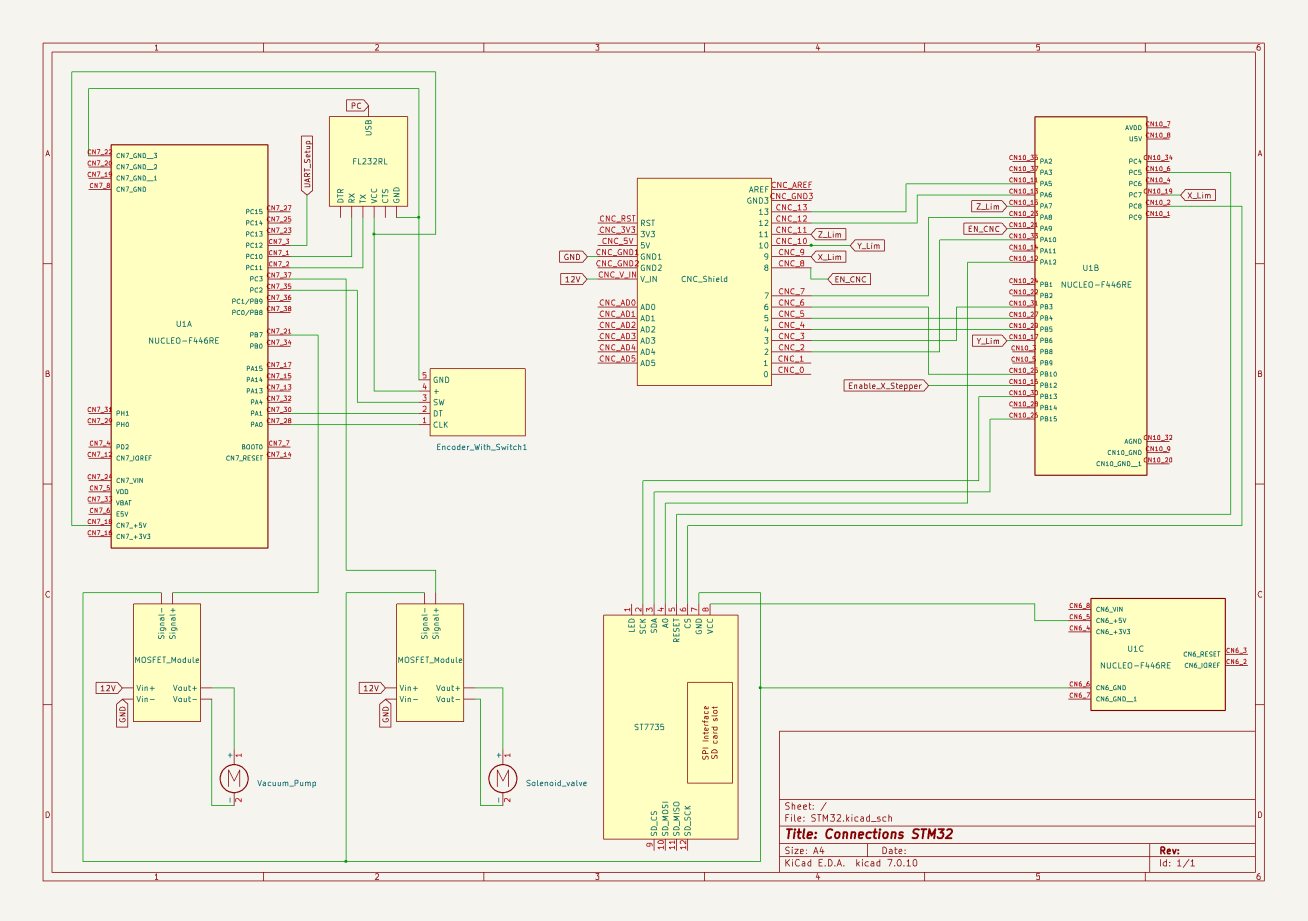

Figure 5.1: Connections between the STM32 chip and the different components

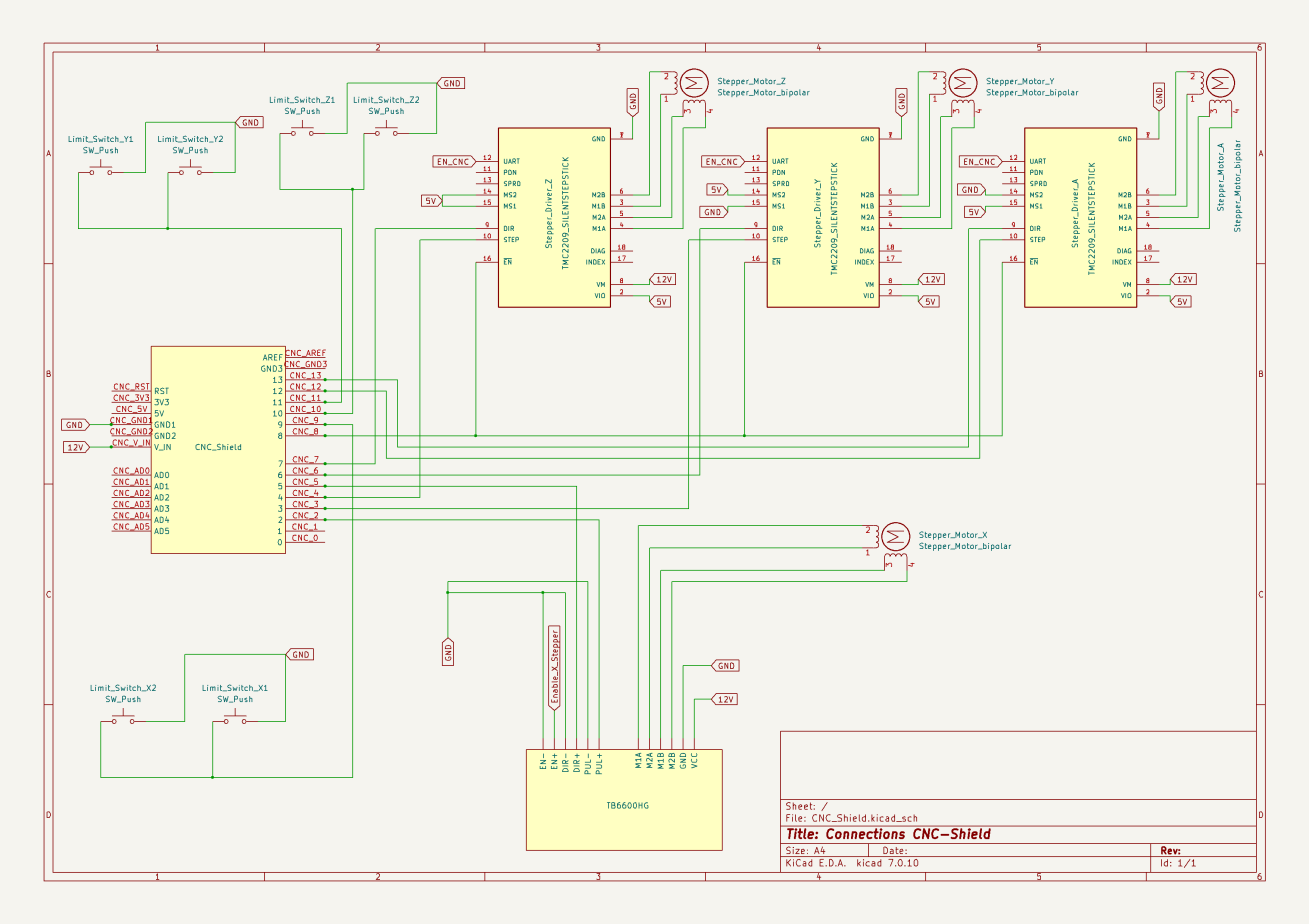

Figure 5.2: Connections between the CNC shield and the different components

5.2 Dataflow

As previously mentioned, the pick-and-place machine will operate with G-code instructions as its input. These G-code instructions, which will be covered in more depth later, will ideally be generated by a separate program using a PCB schematic. This generated G-code file contains a list of commands dictating specific movements and actions needed to fulfil the operation.

Since this list is primarily stored on the PC which runs the G-code generation program, the FT232RL USB TO TTL converter will be used to transmit it to the microcontroller. After this transmission, the G-codes can now be sequentially interpreted by the microcontroller. These interpreted commands are then used by the microcontroller to control the different actuators of the machine. Depending on the specific command, this can be either one of the four stepper motors, the solenoid valve or the vacuum pump.

Controlling the stepper motors happens through the aforementioned stepper motor drivers. The microcontroller simply has to send an amount of steps, or digital pulses, and a direction to the driver and the driver will convert this into two sequenced phases that precisely control the motor. Three of the four drivers together with all three end stop connections are mounted on top of the CNC shield. This shield forms an abstraction layer between the microcontroller and the stepper motor drivers. This abstraction layer decreases the amount of necessary connections and complexity, and subsequentially makes controlling the motors more straightforward.

As the CNC shield does not offer adequate connections to set up all four drivers through a single UART connection, a separate UART connection between the microcontroller and the drivers has to be made. An example of such a connection is displayed in figure 5.3. This connection allows the microcontroller to precisely configure the operating mode of the three stepper motor drivers connected to the CNC shield.

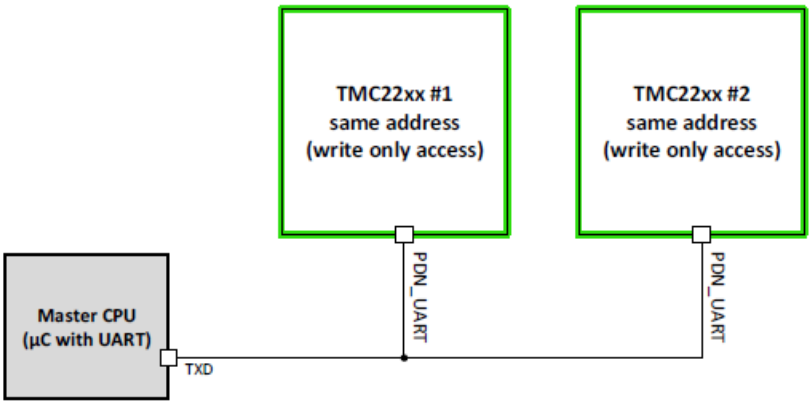

Figure 5.3: UART bus between microcontroller and two TMC2209 drivers [24]

As the X-axis stepper motor driver is not fitted to be integrated with the CNC shield, its connections to the microcontroller are made separately. Even though this controller cannot be controlled through the CNC shield, it will still be connected to the pins reserved for the fourth stepper driver on the CNC shield for convenience purposes.

While the CNC shield offers 6 external end stop connections pins, internally these are mapped to only three pins on the microcontroller. This means that since each axis contains two end stops, their signals will be combined when reading out their status. This is not an ideal situation, as directly determining which end stop is pressed at a given time becomes impossible. Luckily, this problem can be resolved by examining the current status of a machine whenever an end stop is pressed. When an error occurs and the machine hits an end stop, one can assume that the end stop that has been hit is located on the side of the axis that the machine was currently moving towards. Determining which end stop is activated during a homing sequence is not essential, as a homing sequence always happens in the same direction.

The microcontroller also needs to control both the vacuum pump and solenoid valve to pick-and-place different components. As mentioned before, this control will happen through a step-up voltage MOSFET module, since the microcontroller is not able to deliver the operating voltage level of these components.

Throughout the operation of the machine, status information can be displayed in two distinct ways. Simple and routine information can be displayed on the screen attached to the microcontroller, so the user can understand what operation the machine is currently carrying out in the glimpse of an eye. More complex and extensive debug information is sent to the PC using a UART connections. On the PC, it can be visualized in a standard serial terminal to give the user precise updates about the status of the machine. A complete overview of the dataflow mentioned in the previous paragraphs can be seen in the dataflow diagram displayed in figure 5.4 on the next page.

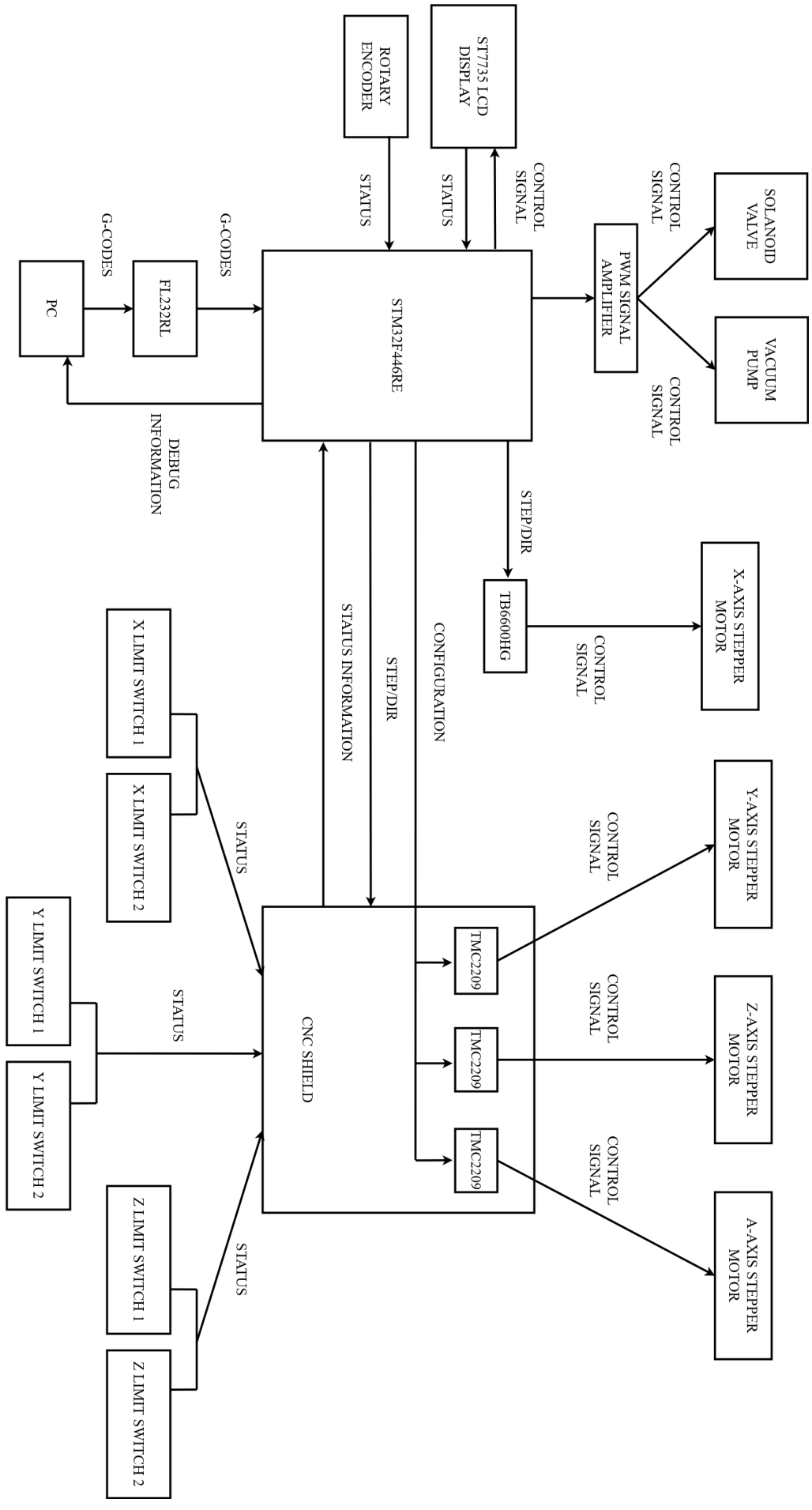

Figure 5.4: Dataflow diagram of the system

Chapter 6

Software architecture

6.1 Software components

6.1.1 IDE

An integrated development environment (IDE) is software that allows the streamlining of the programming process by combining features. Typically, an IDE will include a source code editor, compiler, and a debugger alongside many IDE specific extensions [77]. By combining these features in a single program, the programmer does not need to spend time configuring and learning these different features. Using an IDE also has multiple downsides, like the fact that they are often language specific, for example. However, the time saved by using them and the automation features they offer generally outweigh other considerations.

STM32CubeIDE, from which the interface can be seen in figure 6.1, is an all-in-one multi operating system development tool developed by STMicroelectronics. It is an IDE specifically designed for developing C and C++ applications that can be deployed on a wide range of STM32-based products [78]. As the code for this project will be deployed on the STM32F446RE microcontroller, this IDE offers many benefits and thus will be the selected development environment for this project.

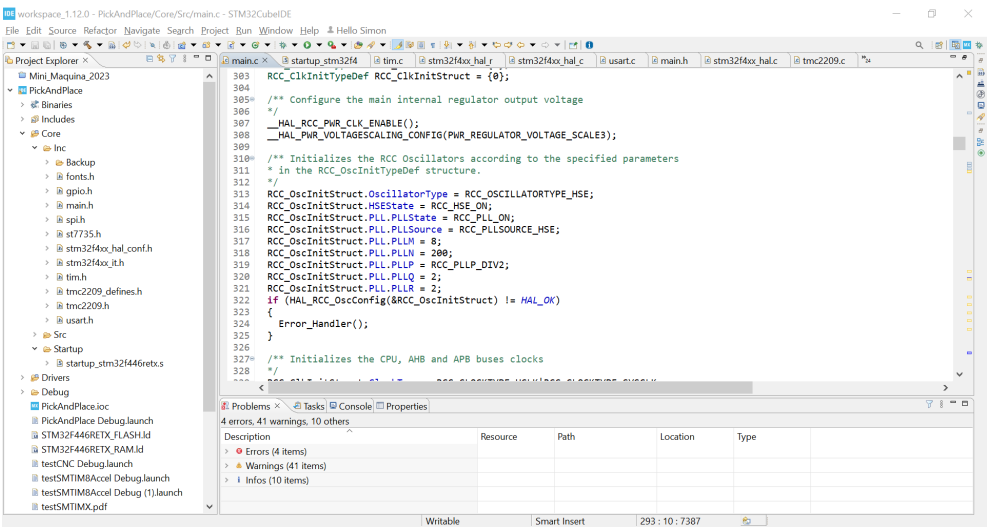

Figure 6.1: STM32CubeIDE

6.1.2 HAL and LL

When developing embedded applications on STM32 microcontroller with STM32CubeIDE, several drivers and software libraries are available to streamline the process. The Hardware Abstraction Layer (HAL) and the Low Layer (LL) drivers are particularly important. Both drivers aim to simplify the programming process by providing ready-to-use, low-level functions. They are important building blocks on which the rest of the STM32CubeIDE ecosystem is based, as can be seen in figure 6.2.

The HAL provides a straightforward, generic set of functions and definitions that interact with the hardware. By using the function in the HAL, developers can perform operations such as reading or writing I/O data or communicating over the serial peripheral interface with a single function call. This abstraction layer therefore drastically speeds up development by eliminating the need to manually configure registers and bits in order to perform low-level operations. Aside from speeding up the development process, using the standardized HAL functions significantly boosts the portability level of applications and hides peripheral complexity from the end-user [79].

The LL provides functions and definitions that operate at register level. These functions and definitions are based on the available features of the STM32 peripherals. Since the LL operates at an even lower level than the HAL, it is able to achieve better optimization at the cost of a lower portability. However, to fully leverage the LL and unlock its optimization potential, an in-depth understanding of the microcontroller and peripherals is essential [79].

The HAL and LL drivers can operate independently or in a mixed mode. In mixed mode, the drivers complement each other, covering a broad range of application requirements.

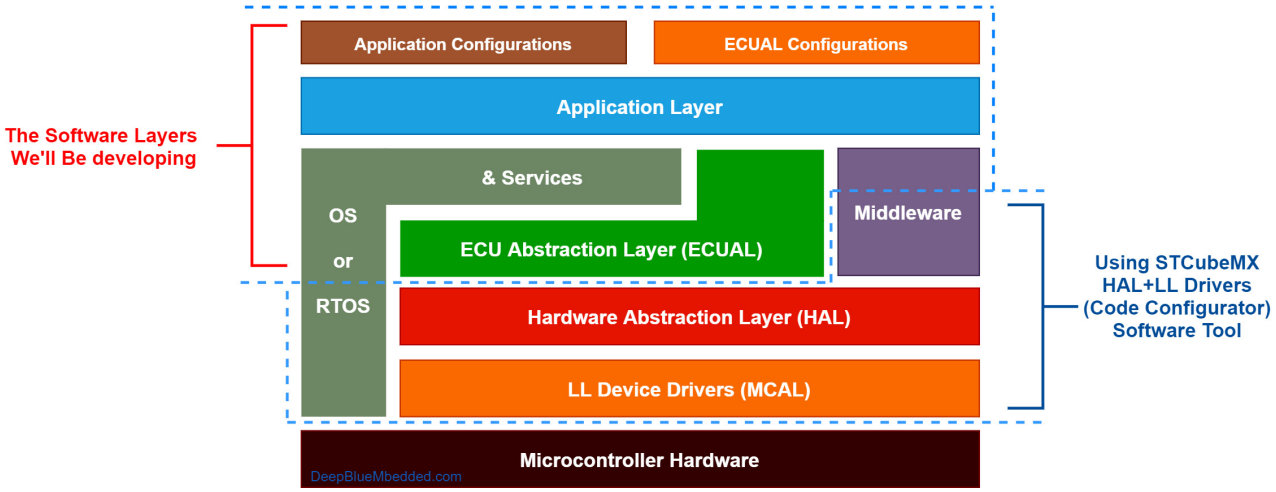

Figure 6.2: Structural layout of STM32CubeIDE [25]

6.1.3 Clocks and timers

Clocks are an essential element of microcontrollers, if not the most essential. The frequency at which they tick determines the speed at which the processor executes instructions. This predetermined execution speed provides a reference time that allows different components of the microcontroller to synchronize their operations [80].

Modern day microcontrollers offer many different clocks that originate from different sources. Broadly, they can be distinguished into high speed and low speed signals that originate from a clock source. a systematic overview of all these different sources and signals is visualized by STM32CubeIDE using a clock configuration tree.

Figure 6.3 illustrates a portion of this tree, highlighting the main clocks used in this project. As shown in the figure, all these clocks originate from the system clock (SYSCLK) ticking at a frequency of 100 MHz. They achieve their respective frequencies through a series of multiplication and division steps.

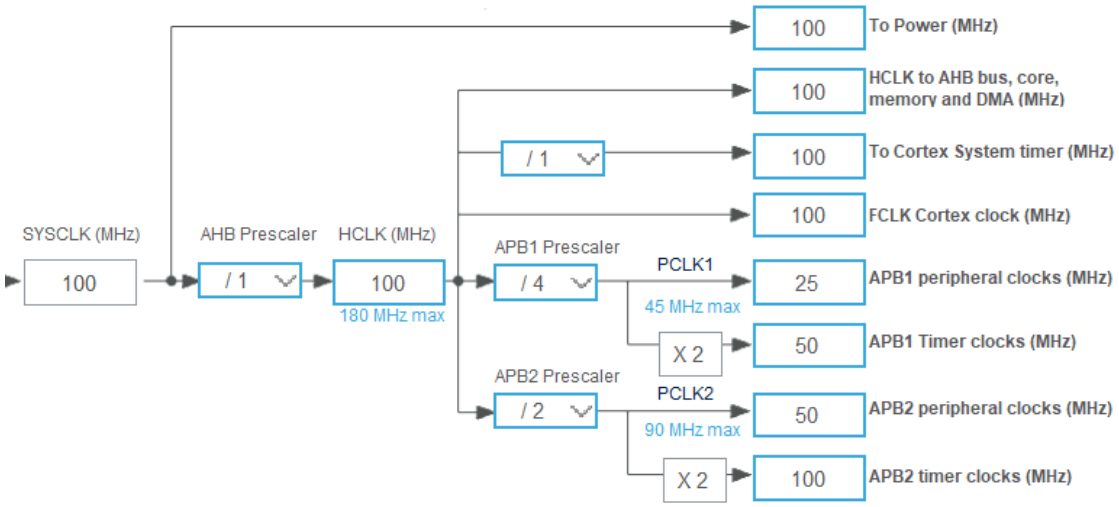

Figure 6.3: Part of the STM32CubeIDE clock configuration tree

It is essential to note that figure 6.3 presents only a segment of the clock configuration tree. A complete clock configuration tree offers many complex and intricate configuration options that can be useful for highly specific applications [81]. The decision to include only a portion of it was deliberate, as the project did not utilize this complex functionality.

For this project, the most important distinction between these clocks is the frequency they run at and the accuracy they can achieve. This distinction is oversimplified, but discussing all the different configurations falls outside the scope of this project.

While the clocks provide many indispensable functions for the operation of the microcontroller, their main use in this project stems from them being the foundation for timers.

Timers are a feature in microcontrollers that allow users to measure the execution time of instructions. In its most basic form, a timer is a digital logic circuit containing a certain number of bits that counts up every clock cycle until its max value is reached, after which it starts over [82]. This number of bits is usually referred to as the counter resolution, as it refers to the maximum count value. For example, a timer with a counter resolution of 16 bits can count from 0 to 65 536 and will thus start over after 65536 clock cycles.

A modern microcontroller contains various different timers, each with their own specific characteristics and set of options that can be modified. The basic speed at which a timer counts is determined by the clock to which it is connected. Since this speed might be too fast for some applications, it can further be altered by modifying the set of options connected to each timer. The most important of these settings will now be discussed briefly.

- Prescaler: The frequency at which the counter changes can be altered by dividing the main clock frequency by a value called the prescaler. For example: when using a timer with a 16 bit counter resolution connected to the main clock with a frequency of 70 MHz, the timer rolls over every 936 microseconds. If the application requires the timing of longer events, a prescaler has to be used. Using a prescaler of 70, this main clock of 70 MHz could effectively be turned into a 1 MHz clock [82].
- Auto reload register: In order to modify the value at which a timer rolls over, we can change the value in the auto reload register. By changing this value, a timer is therefore no longer bound by the maximum value defined by its number of bits [82].

Timers can be used for various purposes like counting pulses, measuring time periods, generating pulse width modulation (PWM) signals, triggering external devices and many more. The STM32 microcontrollers generally contain 3 different kind of timers that each have their own purposes. They can be classified as follows:

- General purpose timers: most timers in STM32 microcontrollers are general purpose timer. They can be used for any timer-counter-related purpose, hence the name [83].
- Basic timers: basic timers don't contain I/O channels for input capture/PWM generation. Therefore, they are strictly used for time-base generation purposes [83].
- Advanced control timers: advanced timers are similar to general purpose timers, but they contain additional features such as generating complementary PWM signals as well as generating brake and dead-time for such signals [83].

A complete overview of all timers available on the STM32F446RE chip and their respective features is displayed in table 6.1.

Table 6.1: All timers available on the STM32F446RE chip and their respective features [1]

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary outputs
Advanced control	TIM1, TIM8	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	3
General-purpose	TIM2, TIM5	32-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM3, TIM4	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM15	16-bit	Up	Any integer between 1 and 65536	Yes	2	1
General-purpose	TIM16, TIM17	16-bit	Up	Any integer between 1 and 65536	Yes	1	1
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No

The main use for timers in this project is generating PWM signals, as these are used to control all four stepper motors. A specific basic or advanced control timer is selected for every motor. Every time this timer reaches its maximum value and rolls over, a step will be sent to the motor. By modifying the prescaler and auto reload register of these timers, the speed of the stepper motor can thus be controlled.

Another application of timers in this project involves tracking the speed and direction of rotation of the rotary encoder. As previously mentioned, an encoder produces two out-of-phase pulses on separate lines. These pulses can be captured by two channels of the same timer on the STM32 microcontroller by configuring the timer's combined channels in encoder mode. In this mode, the timer is synchronized with the external source of the rotating encoder. Setting the auto-reload register to 1 causes the timer to overflow each time the encoder rotates slightly. The frequency of these overflows corresponds to the speed of the encoder's rotation. Additionally, by determining which channel triggers first, the direction of the encoder's rotation can be determined.

6.1.4 GPIO

GPIO, short for general purpose input/output, is a versatile type of pin found on all STM32 microcontrollers. These pins can be configured to serve as either digital or analogue input/output pins. Beyond the basic configuration, their input/output speed, voltage levels, and maximum current and many more characteristics can also be modified [84].

Interacting with these GPIO pins is straightforward, since the HAL driver provides dedicated functions for reading from and writing to these pins.

6.1.5 SPI

SPI, short for Serial Peripheral Interface, is a common communication protocol that enables an embedded device, known as the master, to communicate with external devices, referred to as slaves. This protocol is synchronous, bus-based, and supports full-duplex communication, allowing data to be sent and received simultaneously [26]. It can be configured in various ways, with the most common being the point-to-point connection displayed in figure 6.4.

Figure 6.4: Point-to-point connection over the SPI protocol [26]

This point-to-point configuration uses four specific pins on both devices, with the microcontroller pins labelled as follows:

- Serial clock (SCK): A connection between these two pins on the master and slave device create a separate clock line between the two devices. This enables synchronous communications and allows for high-speed data transfers. It is also known as the clock (CLK) pin [26].
- Serial data out (SDO): This pin on the master device transmits data to the slaves. It is also known as master out slave in (MOSI) [26].
- Serial data in (SDI): This pin on which data transmitted by the slaves is received on the master. It is also known as master in slave out (MISO) [26].
- Chip select (CS): The SPI protocol allows multiple slaves to be connected to its bus. The signal on this pin is used to address the different slaves and differentiate their answers. This pin is also known as slave select (SS) [26].

STM32 microcontrollers feature multiple pins that can be configured to perform these functions, thus enabling multiple SPI communications to be established. This communication protocol is used in the project to establish a point-to-point communication channel between the microcontroller and the diplay. Communicating over this channel is straightforward, as the HAL provides ready to use functions for this purpose.

6.1.6 UART/USART

UART, short for Universal Asynchronous Receiver/Transmitter protocol, refers to the hardware used to facilitate several serial communication protocols. These serial protocols enable full-duplex data exchanges between embedded systems and external devices [27]. As indicated by its name, UART operates asynchronously. However, there is a variant known as USART, Universal Synchronous/Asynchronous Receiver/Transmitter, that supports both synchronous and asynchronous communication.

In synchronous communication over USART, the microcontroller will generate a data clock and transmit it to the external device. In asynchronous communication via USART or UART, both the microcontroller and the external device generate the same data clock internally. Since no clock signal is exchanged between the two parties, both of them must be operating on the same baud rate.

A full-duplex USART connection requires a minimum of two pins on each device: receive data (RX) and transmit data (TX). These pins are enough to facilitate the transmission and reception of serial data in standard UART mode. For synchronous communication, an additional clock pin is necessary to share the clock signal between the devices. This difference in pins and connections is displayed in figure 6.5.

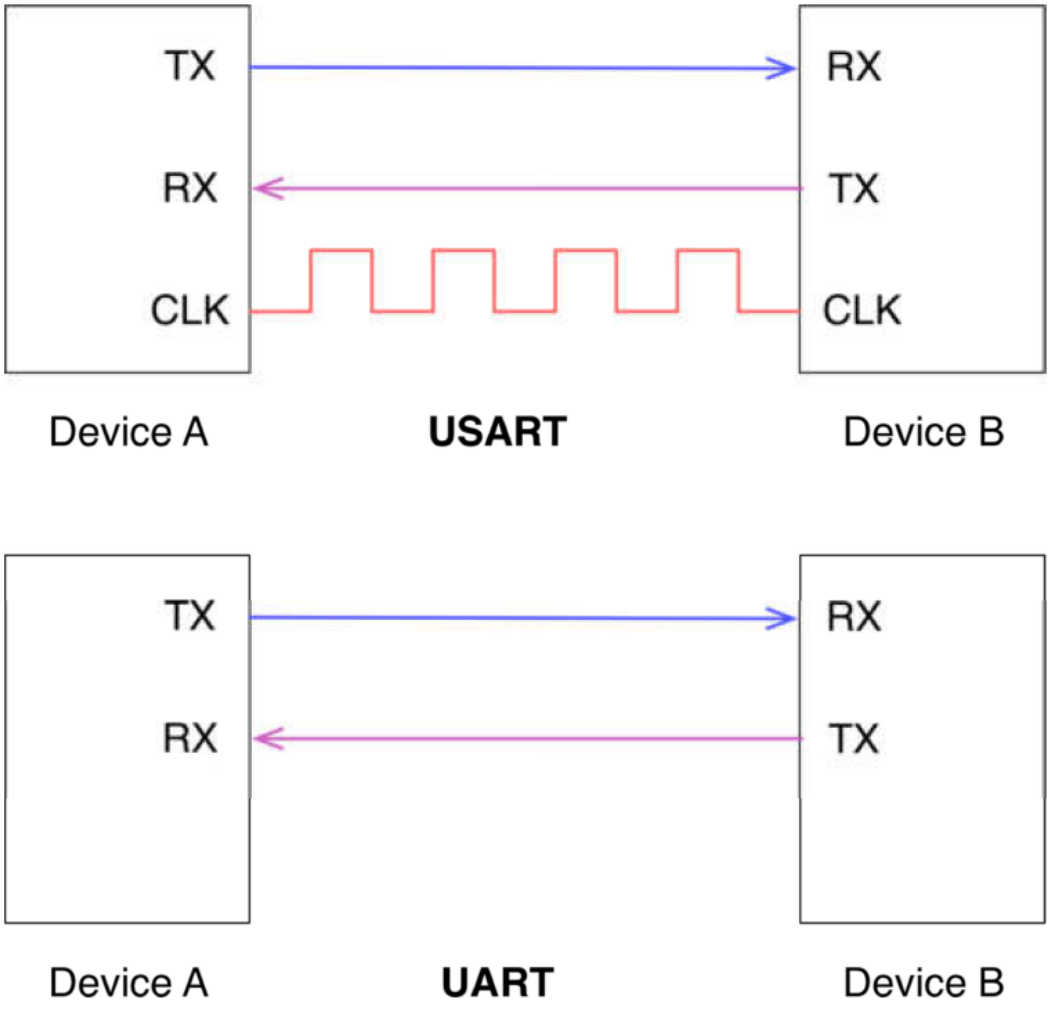

Figure 6.5: Difference in connections between UART and USART [27]

While establishing a USART communication channel between two devices requires fewer pins than an SPI channel, the data transfer speed over SPI is generally higher than that of USART [85].

Both UART and USART are employed in this project to establish multiple communication channels. The first channel will be used for transmitting debug information between the microcontroller and the PC. This is done using a USART communication channel with a specified baud rate of 9600. The second channel will be used for transmitting the G-code instructions from the PC to the microcontroller. This is done using a USART communication channel between the PC and the FT232RL module, which will forward the information to the microcontroller. This channel has a specified baud rate of 115200. Lastly, a third channel is established between the microcontroller and the three TMC2209 stepper motor drivers to configure the operation mode of these drivers. This will be done using the UART bus displayed in figure 4.3 at a specified baud rate of 9600.

6.1.7 G-codes

While so far G-codes were mentioned to be the commands that dictate the machine to make specific movements and do actions, this is an oversimplification in some regards. G-code is short for geometric code and is a type of programming language primarily used in the field of CNC machines. It does not only tell the machine where to move, but also at what speed, how long to wait in between movements, what unit of distance to use and many other things [86]. While the term G-code often refers to the programming language as a whole, it is also only one of the two types of commands used.

General command lines are responsible for moving the machine. These commands are identified by the letter 'G', as in G-codes. The second type of commands, miscellaneous command, generally instruct the machine to perform non-movement tasks. These tasks can include starting and stopping the machine, changing tool heads, ... These commands are identified with the letter 'M', and are therefore also called M-codes [86].

While G-codes are largely standardized across different machines, the variation in tasks performed by these machines often calls for the use of unique G-codes. This need for customization has led to the development of multiple G-code 'flavours' that differ slightly from one another.

While a wide range of open source G-code interpreters and CNC controllers are available online (e.g. [87]) for this specific project, the choice was made to develop our own interpreter. This approach provides maximum implementation flexibility and allows future students working with this project to start exploring the inner working of G-codes without facing the steep learning curve of the rather complex standardized interpreters. Another reason for this design choice is that since we are working with a prototype, the machine will accept only the essential G-codes needed for operation. The list of G-codes used in this project includes the following:

- G1 [Xpos] [Ypos] [Zpos]: Instructs the machine to linearly move to the X, Y and Z coordinates in the instruction.
- G4 [Milliseconds]: Instructs the machine to wait a certain amount of milliseconds before resuming operation.
- G20: Changes the unit in which the machine operates to mil. 1 mil is 0.02 millimetres and the control over the machine is therefore more precise.
- G21: Changes the unit in which the machine operates to millimetres.
- G28: Indicates a homing sequence and instructs the motors to move until an end stop is hit
- M3: Instructs the machine to activate the vacuum pump and solenoid valve in order to pick up a component
- M5: Instructs the machine to deactivate the vacuum pump and solenoid valve in order to release a component
- M6: Instructs the machine to move towards the nozzle holder and attach a nozzle on the tool head. This will also home the motors afterwards.
- M7 Instructs the machine to move towards the nozzle holder and deposit the currently employed nozzle. This will also home the motors afterwards.

6.2 Pinout configuration

The STM32F446RE microcontroller used in this project offers 76 digital pins which can be configured to preform the various functions mentioned throughout this chapter. This configuration can be done using the STM32 CubeIDE and a summary of the utilized pins and their functions can be seen in figure 6.6 and table 6.2.

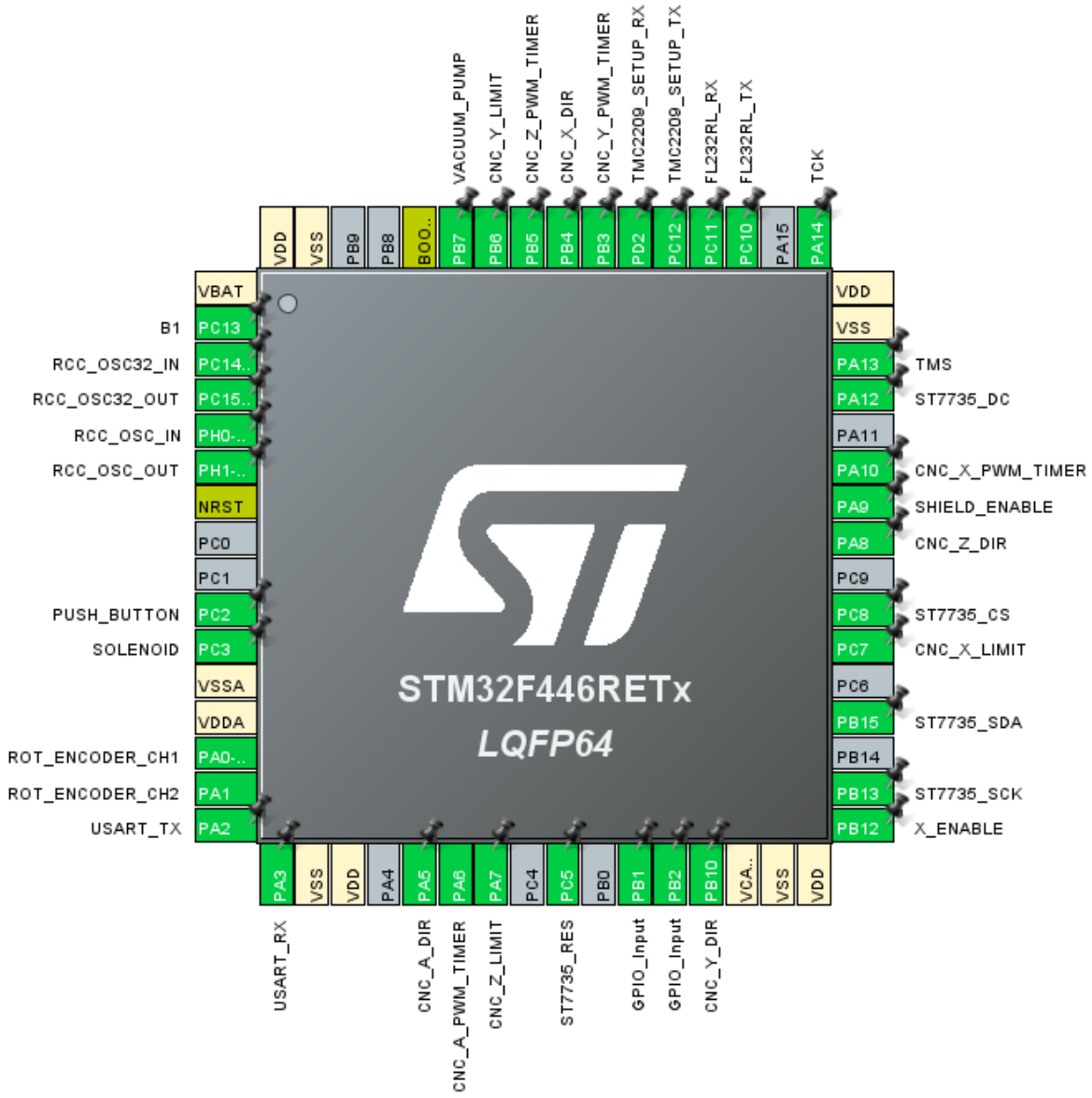

Figure 6.6: Pinout configuration of the STM32F446RE chip

Table 6.2: Table wise pinout configuration of the STM32F446RE chip

Component name	Component pin name	STM32 pin	Pin label CubeIDE	Pin function CubeIDE
TB6600HG (X-axis stepper motor driver)	ENABLE+	PB4	CNC_X_DIR	GPIO_Output
TB6600HG (X-axis stepper motor driver)	PULSE +	PA10	CNC_X_PWM_TIMER	TIM1_CH3 PWM Generation
TB6600HG (X-axis stepper motor driver)	DIR+	PB12	X_ENABLE	GPIO_Output
CNC Shield	Y.DIR	PB10	CNC_Y_DIR	GPIO_Output
CNC Shield	Y.STEP	PB3	CNC_Y_PWM_TIMER	TIM2_CH2 PWM Generation
CNC Shield	Z.DIR	PA8	CNC_Z_DIR	GPIO_Output
CNC Shield	Z.STEP	PB5	CNC_Z_PWM_TIMER	TIM3_CH2 PWM Generation
CNC Shield	A.DIR	PA5	CNC_A_DIR	GPIO_Output
CNC Shield	A.STEP	PA6	CNC_A_PWM_TIMER	TIM13_CH1 PWM Generation
CNC Shield	END STOP X	PC7	CNC_X_LIMIT	GPIO_Input
CNC Shield	END STOP Y	PB6	CNC_Y_LIMIT	GPIO_Input
CNC Shield	END STOP Z	PA7	CNC_Z_LIMIT	GPIO_Input
CNC Shield	SpnEN	PA9	SHIELD_ENABLE	GPIO_Output
Rotary encoder with switch	DT	PA0	ROT_ENCODER_CH1	TIM5_CH1 Encoder mode
Rotary encoder with switch	CLK	PA1	ROT_ENCODER_CH2	TIM5_CH2 Encoder mode
Rotary encoder with switch	SW	PC2	PUSH_BUTTON	GPIO_Input
ST7735	CS	PC8	ST7735_CS	GPIO_Output
ST7735	Reset	PC5	ST7735_RES	GPIO_Output
ST7735	DC	PA12	ST7735_DC	GPIO_Output
ST7735	SDA	PB15	ST7735_SDA	SPI2_MOSI
ST7735	SCK	PB13	ST7735_SCK	SPI2_SCK
FL232RL	RX	PC10	FL232RL_TX	USART3_TX
FL232RL	TX	PC11	FL232RL_RX	USART3_RX
TMC2209	RX	PC12	TMC2209_SETUP_TX	UART5_TX
Vacuum Pump	+	PB7	VACUUM_PUMP	GPIO_Output
Solenoid Valve	+	PC3	SOLENOID	GPIO_Output

6.3 General outline of the code

The codebase for this project was entirely written in the C programming language. The working of the code is relatively straightforward. After defining necessary variables, functions and structures, a main function orchestrates the machine's operation. It begins by initializing all configured peripherals, configuring the system clock and evoking the driver setup. After this initialization process, it enters an indefinite while loop responsible for controlling and processing user interaction via the display. A choice menu provides options for either motor testing or machine operation. Once an option is selected, various functions, which will be elaborated on later, are utilized to process the user's commands and operate the machine accordingly.

While the main codebase is original, several libraries were incorporated to streamline development.

In order to interact with the display, a STM32 HAL-based library was employed. This library was sourced from GitHub and developed by user "Afiskon" [88]. It contains various functions and fonts specifically developed for use on the ST7735 driver for this display.

For communication with the TMC2209 stepper motor drivers, a SMT32 HAL-based library was employed. This library was sourced from GitHub and developed by "Veysi Adin" [89]. It contains all essential functions for seamless interaction with these drivers.

In addition, a variety of functions and libraries developed by STMicroelectronics were used (e.g. aforementioned [79]). These functions and libraries made it easier to control all aspects of the microcontroller.

Lastly, fragments of code authored by students at the UPV, working on similar projects, have been consulted. However, no direct copying of this code has been done, and it was mostly used for inspiration purposes.

6.4 Overview of employed functions

- Visual functions: These functions present information to the user by displaying it on the screen connected to the microcontroller.
  - showHomeScreen(): This function initializes the ST7735 driver of the screen and displays the initial home screen.
  - showTestScreen(): This function displays the choice menu after the user has selected the test option on the initial home screen.
  - interactHomeScreen(): Handles the interaction between the user and the choice menu presented on the home screen. By reading and processing the encoder values, it allows the user to select and confirm the desired option.
  - interactTestScreen(): This function provides similar functionality to one previously discussed but for the test screen.
- Movement functions: All the functions needed to move the machine to the desired position.
  - void sendStepsX(uint32_t steps, uint8_t direction): This function controls the movement of the X-axis stepper motor. In order to accomplish this, a specified number of steps and a direction have to be provided as input parameters. Depending on the specified direction, it sets or resets the direction pin of the X-axis stepper motor driver. The function then starts the timer connected to the step pin of this X-axis stepper motor driver. This timer will employ pulse-width modulation in order to generate pulses that will be driving the motor.

    During the movement process, the function continuously checks the status of an end-flag that indicates when the movement is done. It also monitors the status of the X-axis end stop, of which the activation can mean two things. If the end stop is triggered while a homeFlag variable is activated, it indicates a homing sequence. If this is the case, the function retracts the motor by 50 steps from the end stop and establishes the zero position. Otherwise, if the end stop is triggered during normal operation without the homeFlag variable activated, the machine exceeded its physical limits and the operation will exit.

    This function also includes input debouncing to ensure that no false presses due to noise or electromagnetic interference occur. Practically, this is implemented by checking the state of the button twice with a 10-millisecond delay in between. Only if the push button is pressed during both checks will the function register it as a true button press.
  - void sendStepsY(uint32_t steps, uint8_t direction): This function serves the same purpose as the one previously described but for the Y-axis stepper motor.
  - void sendStepsZ(uint32_t steps, uint8_t direction): This function serves the same purpose as the one previously described but for the Z-axis stepper motor.
  - void sendStepsA(uint32_t steps, uint8_t direction): This function serves the same purpose as the one previously described but for the A-axis stepper motor.
  - void moveStepperMotorX(uint32_t steps, uint8_t direct): This function controls the movement profile of the stepper motor along the X-axis. This function is designed to

ensure smooth and precise motion by employing hardcoded speed profiles.

For longer movements, where the number of steps is 400 or more, the function divides the movement into three phases: acceleration, constant speed and deceleration. By utilizing this profile, the speed follows a trapezoidal pattern over time. This linear increasing and decreasing of the speed before and after operating at maximum speed is crucial when working with stepper motors. Starting stepper motors at maximum speed would result in it skipping steps and therefore negatively impacting the accuracy and precision of the movement [90]. The duration and speed characteristics of these phases are completely managed using variables and can therefore easily be changed.

For shorter movements, where the number of steps is less than 400, the function uses a simplified speed profile with three levels.

In both cases, the variable containing the global position of the motor is updated after the movement.

The increasing and decreasing of the speed is done by updating the auto reload register of the timer responsible for pulse width modulation generation. The control of these timers, is done by the aforementioned sendStepsX function.

- void moveStepperMotorY(uint32_t steps, uint8_t direct): This function serves the same purpose as the one previously described. However, the speed profile has been adjusted to optimally control the Y-axis stepper motor.
- void moveStepperMotorZ(uint32_t steps, uint8_t direct): This function serves the same purpose as the one previously described. However, the speed profile has been adjusted to optimally control the Z-axis stepper motor.
- void rotateStepperMotorA(uint32_t degrees): This function controls the movement of the rotary A-axis stepper motor. It takes in degrees as input and transforms this into a discrete number of steps.
- void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim): This function, provided by the previously discussed hardware abstraction layer, is called whenever a timer triggers an interrupt. In this project, each interrupt corresponds to a step, allowing this function to track the status of stepper motor movement and signal when the movement is complete.
- void home(): This function commands the stepper motors of all three movement axes to return to their starting positions. It achieves this by setting the homeFlag to 1 and continuously moving all motors in a specified direction until the end stops are triggered. Once all motors reach their respective end stops, the variables containing the current X, Y, and Z positions are reset. Calling this function at the beginning of operation establishes a standardized starting position, ensuring precise and accurate movements.
- void pickUpComponent(): This function enables the vacuum pump and opens the solenoid valve. This creates a vacuum used by the machine in order to pick up a component.
- void putDownComponent(): This function disables the vacuum pump and closes the solenoid valve. This will release the vacuum and therefore place the component.

  - void employNozzle(): This functions moves the machine so it attaches a nozzle currently present in the holder. For this, hardcoded specific instructions are used. The machine will also do a homing sequence at the end of this function.
  - void depositNozzle(): This functions moves the machine so it deposits the currently employed nozzle in the holder. For this, hardcoded specific instructions are used. The machine will also do a homing sequence at the end of this function.
- Debugging functions: Used to deliver relevant system information to the user.
  - void printUsart2(char text[]): This functions transfers an array of chars to the operating laptop using the established UART2 channel.
  - void putCharUsart2(char ch2[1]): This functions transfers a single char to the operating laptop using the established UART2 channel.
- Helper functions: Used for various objectives related to the working of the pick-and-place machine.
  - void setupDrivers(void): This function manages the setup of the TMC2209 stepper motor drivers. This setup process is done via a UART channel, allowing full control over all driver settings. It initializes key parameters, including microsteps per step, pulse width modulation frequency, and the StealthChop and CoolStep features.
  - o uint8_t pushButton(void): This function checks whether the push button, implemented as a switch on the rotary encoder, is currently pressed or not. This function also contains an input debouncing routine for aforementioned reasons.
  - uint8_t readNumber(char charBuffer[], uint8_t charCounter, float *floatPtr): This function extracts a floating-point value from a string. This functionality is needed as the G-codes strings need to be converted into numerical values. The code is loosely based on the avr-libc strtod() function by "Michael Stumpf" and "Dmitry Xmelkov" [91], as well as many freely available conversion method examples.
  - uint32_t millimeterStepConversion(double millimetres, uint8_t motor): This functions converts millimetres into a discrete number of steps for the different stepper motors. This is necessary since the number of steps per millimetre varies significantly between different motors. Therefore, millimetres is the standardized unit of operation for CNC machines.
  - uint32_t milStepConversion(double mil, uint8_t motor): This functions converts mil into a discrete number of steps for the different stepper motors. This is necessary since the number of steps per millimetre varies significantly between different motors. This function is only executed when the machine is operating in mil units and the X-axis stepper motor driver is set to a microstepping option of 1/16. This is necessary since this function provides fine control over the motor movements and can only be accurate is the smallest step any of the motors can make is equal or less than 1 mil.
- General functions
  - void work(): This function manages the processing of G-code commands to direct the machine's operations. It starts off by continuously listening for incoming G-code commands via UART communication. As each character of the G-code command is received, it is stored in a buffer until the end of the command is detected (denoted by

the '/' symbol).

Upon receiving the complete list of G-code commands, the function proceeds to iteratively parse each command individually. To do this, it extracts the command letter and numerical value and determines what actions need to be taken based on these parameters.

Subsequently, the function executes the desired actions by invoking corresponding functions for each of the received commands. It continues this process until all received commands have been processed, and all corresponding functions have been executed.

In case an error is detected at any stage throughout this process, the machine will display an appropriate error message and prematurely exit the operation.

Chapter 7

Test setup

7.1 Final machine setup

The machine displayed in figure 7.1 is the final product of this project. The finished machine can now be tested in order to ensure that it functions as expected and meets the accuracy requirements.

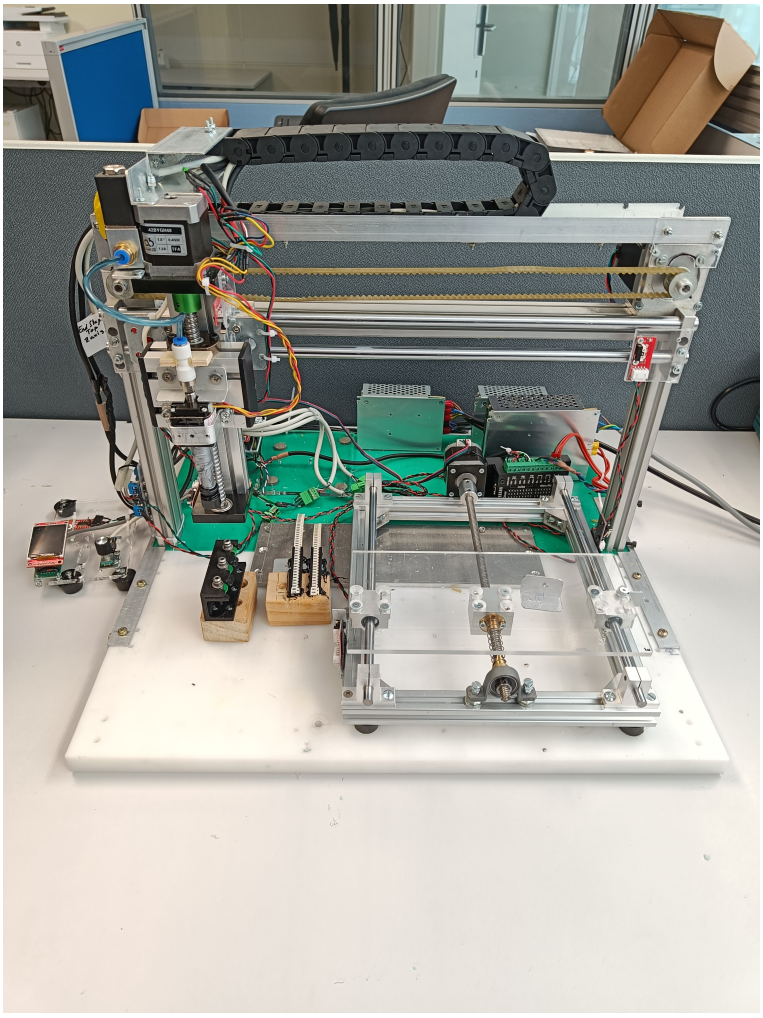

Figure 7.1: Final constructed pick-and-place machine

7.2 Pre-test driver calculations and configurations

Before testing the machine, it is essential to optimally configure the microstepping values of the drivers. While utilizing microstepping increases smoothness, ideally, it will not be used as it reduces the incremental torque of the stepper motor. To determine whether utilizing microstepping is necessary, the accuracy and precision of the current setup without microstepping must be assessed.

Each step represents the smallest possible movement a stepper motor can make. Therefore, the machine's accuracy is directly tied to the size of these steps. In order to judge and compare these sizes, it is crucial to determine how many millimetres of linear movement each step corresponds to. While this conversion is sometimes provided in the stepper motor datasheet, it often needs to be calculated due to its dependence on the specific ball screw or belt drive used. This conversion can be obtained in two ways:

- By measuring the linear movement of the load on the motor over a set number of steps, this value can be practically obtained.
- By inserting all relevant information about the stepper motor and belt drive or ball screw used into a specific formula, this value can be theoretically obtained.

For this project, a practical approach was chosen, as not all relevant information for the theoretical approach was known and had to be measured itself. The measurement process yielded the following millimetres-per-step values:

- X-axis: 0.33 mm/step
- Y-axis: 0.004 mm/step
- Z-axis: 0.02 mm/step

These results clearly indicate that the millimetres-per-step value for the X-axis stepper motor and the respective belt drive is a limiting factor for the machine's accuracy. This bottleneck can be mitigated by setting the driver of the X-axis motor to a microstepping mode of 1/8 or 1/16. However, the practical feasibility of this solution has to be tested in order to make sure that the motor can still deliver adequate torque to move the load while operating in this mode.

In contrast, the millimetres-per-step values for the Y and Z axes provide sufficient precision and accuracy for the machine's intended purposes. Therefore, no microstepping is required for these axes.

7.3 Test setup

In order to achieve full functionality, the machine should be able to pick-and-place components of various sizes using one of the three available Juki nozzles. It also should be able to automatically switch between these nozzles.

In order to verify these capabilities, all these operations have to be tested. Therefore, the following four tests will be executed in isolation:

- Move to the location of a specific component, pick it up and place it on a specified location.
- Move to the location of the nozzle holder and deposit the currently used nozzle.
- Move to the location of the nozzle holder and attach a nozzle present in this holder.
- Pick up a component and rotate it by a specified angle.

Following these isolated tests, the necessary operations will be tested in combination. This will be done by having the machine perform an actual pick-and-place task on the PCB depicted in Figure 7.2. This PCB contains a schematic of a simple oscillator, which can be used to produce a periodic electronic signal. This PCB consists of four components:

- Two capacitors of 10nF and 100nF, both with dimensions of 3.0 × 1.5 mm.
- A resistor of 1MOhm with a dimension of 1.5 × 0.8 mm.
- An LM555 integrated circuit with a dimension of 3.0 × 3.0 mm.

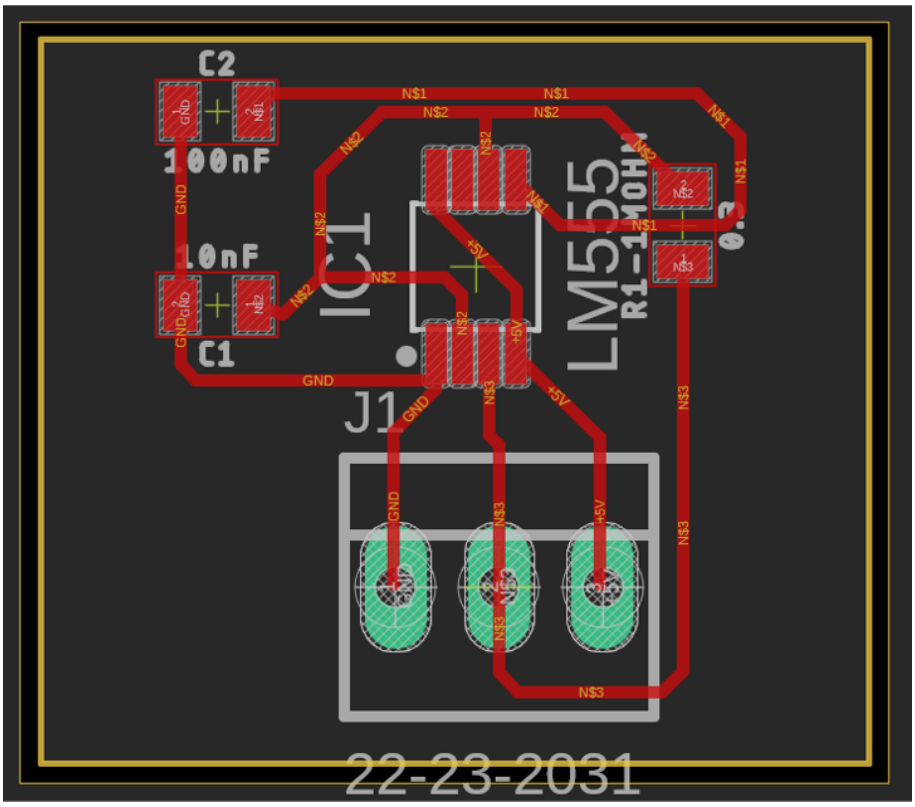

Figure 7.2: PCB of oscillator circuit

Chapter 8

Result

8.1 General results

The final machine presented in this thesis works as expected. The software and electrical systems function correctly, allowing the machine to interpret and execute G-code commands accurately. The head of the machine moves according to the movement profiles and stops when an end stop is hit. All communication channels are operational. System information is correctly displayed to the user, both in a terminal, as shown in Figure 8.1, and on the screen connected to the microcontroller.

With a microstepping mode of 1/16 on the X-axis stepper motor driver, the machine achieves a theoretical accuracy of 0.02 mm. This level of accuracy is sufficient for picking all commonly used components. However, practical factors such as skipped steps and imperfect alignment of the Z-axis lead to inconsistent precision. Consequently, the actual accuracy of the machine is estimated to be around 0.08 mm, which remains adequate for most pick-and-place applications [35]. While operating in this microstepping mode, the X-axis motor can deliver adequate torque for operation.

The machine consistently succeeds in executing the isolated tasks proposed in the previous section. Due to some unexpected delays, the full pick-and-place test could not be completed in time for inclusion in this thesis. Nevertheless, the machine successfully picked up all three components from the task, rotated them, and placed them at the desired positions. Therefore, it is confidently expected that the machine will perform well in the full pick-and-place task.

Additionally, the machine was tested on other small-scale pick-and-place tasks that combined different operations. The output from one of these tests, displayed on the serial terminal of the PC, is shown in Figure 8.1. While these tests did not directly assess the machine's placement speed, they provided an estimation. In its current state, the machine's placement speed is estimated to be around 10 components per minute.

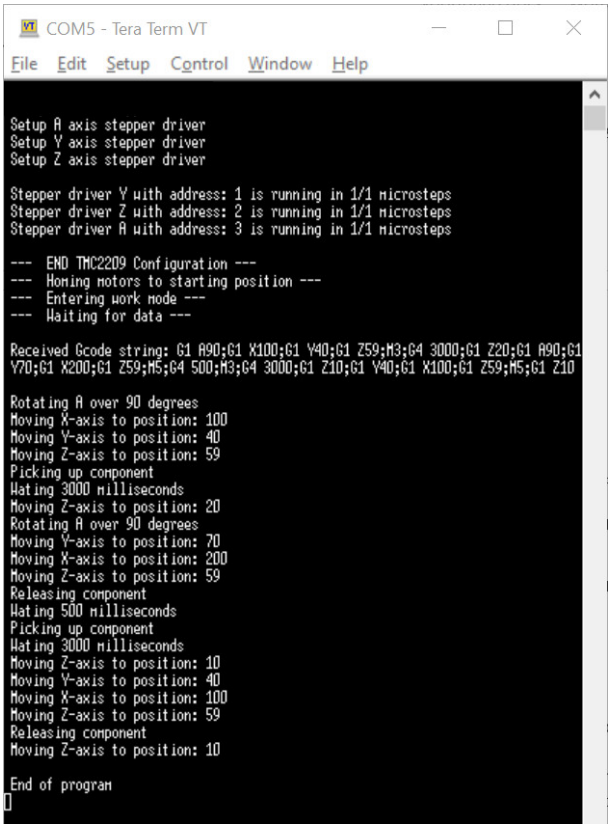

Figure 8.1: Output of system operation communicated over UART

8.2 Financial breakdown

One of the primary objectives of this thesis was to design and implement a cost-effective pick-and-place machine. As highlighted in the component selection section, the machine was constructed using a mix of repurposed components, which were no longer in use, and newly ordered components.

To gain insight into the cost-effectiveness of the presented machine, it is essential to consider and compile the cost of all components. However, creating a perfectly accurate financial breakdown is challenging for several reasons.

First, the price of components can vary significantly between different retailers and over time, especially when ordering online. Second, it is difficult to find accurate price data for repurposed parts that are no longer sold. This issue also arises with components that are typically sold in bulk, such as screws and bolts.

Consequently, the prices listed in Table 8.1 represent average prices derived from various retailers. For components that were no longer available, the prices of similar alternatives were used. To account for potential undervaluation, a maximum added cost was included for components that were not or inaccurately priced in.

Table 8.1 shows that the alternative, non-hierarchical design and carefully considered selection of components resulted in a machine with a total cost of less than 800 euro. However, the presented machine is still a prototype and lacks certain components found in commercial alternatives, such as a vision system and feeder reels. Even when including the cost of these additional components, the machine remains under 1000 euros.

Table 8.1: Financial breakdown of presented pick-and-place machine

Name of the part	price of the part (€)
T-slot profiles	30
Z-axis ball screw + guide rails	90
Y-axis ball screw + guide rails	75
X-axis belt drive + guide rails	30
Cable carrier	15
Yuki nozzles	6
Shielded cable	7
Various connector pieces	40
X-axis motor	31
Y-axis motor	20
Z-axis motor	20
A-axis motor	14
TMC2209 driver x 3	21
TB6600HG driver	20
STM32 NUCLEO-F446RE	24
CNC shield	8
Limit switch x 6	7
ST7735 display	7
Rotary encoder with switch	3
RL232 USB TO TTL	4
Vacuum pump	14
Solenoid valve	9
Mosfet module x 2	3
Power supply CNC shield	16
Power supply X-axis motor	13
Power supply vacuum and solanoid	13
Arduino cables	10
Maximum estimated cost of used components not included in this list	200
Total cost	750

Chapter 9

Proposed improvements

Prior to starting this project, my knowledge in both electrical and mechanical engineering was limited. Additionally, this project was my first experience with the majority of the components used. Consequently, several improvements can be made to optimize and enhance the performance and design of this machine.

There were few reference designs available during the design process due to the alternative structure of the machine. Consequently, some impracticalities are present in the design, such as awkwardly placed components, poorly managed wires and a relatively small operation surface. Additionally, the Z-axis is not perfectly aligned and this negatively impacts the accuracy of the machine. The mechanical design of the machine can therefore certainly be refined.

Utilizing repurposed components that were no longer in use ensured sustainability but proved to be suboptimal in a few cases. While the final machine fulfils its intended purpose, a better balance between component performances could have been achieved. Some components are underpowered compared to others, creating performance bottlenecks. An unrestricted selection of components could have prevented this issue.

The presented machine can be further improved by adding a vision system, which would allow it to calibrate itself without the need for home switches. This would enable continuous calibration during operation, thereby increasing both placement speed and accuracy.

Further optimization of the speed and accuracy is achievable by addressing these bottlenecks currently present in the machine. While this optimization is relatively straightforward and builds upon the knowledge presented in this thesis, time and component restraints prevented its implementation.

Lastly, the codebase, particularly the G-code interpreter, can also be improved. Currently, the machine operates on a limited set of very basic G-code instructions, as non more were required to test the machine. However, with the addition of extra components, more complex G-codes will be necessary. These additional G-codes can be easily integrated into the interpreter due to its flexible design.

Chapter 10

Conclusion

The prototype four-axis pick-and-place machine developed in this thesis demonstrates the feasibility of a non-hierarchical design. Preliminary tests indicate that this alternative design can achieve reasonable accuracy and placement speed, though more extensive testing is needed to confirm these results.

In its current state, the pick-and-place machine presented in this thesis cannot match the performance of commercial models. However, it can compete in terms of cost. Even after including all the components that are missing compared to its commercial counterparts, the total price of the machine remains under 1000 euros. This renders it cheaper than other available options and shows considerable promise for further research.

Additional research will determine whether addressing the current performance bottlenecks and including the missing components can make this machine competitive with commercial alternatives in all categories. However, constructing a machine competitive with current market options was never the primary objective of this project. Therefore, this project is considered a success, as the final machine is a cost-effective proof of concept that can serve as a foundation for further development.

The development and implementation process of this pick-and-place machine required a broad range of knowledge from various engineering fields. Regardless of its outcome, the project has been an extremely valuable learning experience. It is my hope that this machine will serve as an equally valuable knowledge base for future engineering students at the UPV.

Bibliography

[1] STMicroelectronics, "Stm32f446xc/e." Available at https://www.st.com/resource/en/datasheet/stm32f446re.pdf (09/06/2024).

[2] YouTube, "Neoden4(tm4120v) pick and place machine in production." Available at https://i.ytimg.com/vi/Dt7QsoH-SCk/maxresdefault.jpg (10/06/2024).

[3] Otalum, "T slot aluminium profile." Available at https://www.otalum.com/products/t-slot-aluminum-profile.html (10/06/2024).

[4] Heason, "How do ball screws work?." Available at https://www.heason.com/news-media/technical-blog-archive/how-do-ball-screws-work- (10/06/2024).

[5] Indiamart, "Belt drive." Available at https://www.indiamart.com/proddetail/belt-drive-11668903997.html (10/06/2024).

[6] amazon, "Cnccanen rm1204 - tornillo de bola con guía lineal bk10/bf10." Available at https://www.amazon.com.mx/CNCCANEN-RM1204-Tornillo-SBR12UU-m

[7] emaselectric, "15x15mm r:25mm open plastic cable carrier." Available at https://www.emaselectric.com/products/r–cable-carriers/hkp–series/hkp015015r1a–15x15mm-r-25mm-cable-carrier (10/06/2024).

[8] QY-SMT, "Nozzle 500." Available at https://www.qy-smt.com/shop/40011046-nozzle-500-147343attr= (10/06/2024).

[9] mafe72, "Pick and place juki nozzle holder base for chmt36." Available at https://www.thingiverse.com/thing:3527421/files (10/06/2024).

[10] G. Alleman, "Is there an underlying theory of software project management? (a critique of the transformational and normative views of project management)," 10 2002.

[11] Deltaprintr, "Vacuum pump (12v)." Available at https://www.deltaprintr.com/product/vacuum-pump-12v/ (09/06/2024).

[12] Walmart, "1/8" npt dc 12v electric solenoid valve 2 way water air valve normally closed." Available at https://www.walmart.com/ip/1-8-NPT-DC-12V-Electric-Solenoid-Valve-2-Way-Water-Air-Valve-Normally-Closed/533897280 (09/06/2024).

[13] Boris, "How does stepper motor driver work? – the complete explanation." Available at https://blog.poscope.com/stepper-motor-driver/ (09/06/2024).

[14] Automate, "What is the difference between full-stepping, the half-stepping, and the micro-drive?." Available at https://www.automate.org/motion-control/case-studies/what-is-the-difference-between-full-stepping-the-half-stepping-and-the-micro-drive (10/06/2024).

[15] 3Dbro, "Bigtreetech tmc2209 v1.2 stepper driver." Available at https://3dbro.com.au/product/btt-tmc2209-v1-2-stepper-driver/ (10/06/2024).

[16] Novellus, "Raspberry pi, python, and a tb6600 stepper motor driver." Available at https://www.instructables.com/Raspberry-Pi-Python-and-a-TB6600-Stepper-Motor-Dri/ (10/06/2024).

[17] DIY Projects, "Arduino cnc shield version 3.0 with grbl v0.9." Available at http://diyprojects.eu/arduino-cnc-shield-version-3-0-with-grbl-v0-9/ (10/06/2024).

[18] Ebay, "Mechanical endstop for reprap ramps 1.4 3d printer." Available at https://www.ebay.es/itm/222228445421 (10/06/2024).

[19] Indiamart, "Normal 1.8 inch tft lcd module 128 x 160 with 4 io, red." Available at https://www.indiamart.com/proddetail/1-8-inch-tft-lcd-module-128-x-160-with-4-io-2852889863891.html (10/06/2024).

[20] Addicore, "Rotary encoder with push switch." Available at https://www.addicore.com/products/rotary-encoder-with-push-switch (10/06/2024).

[21] Rajguru Electronics, "Ft232rl usb to ttl 5v 3.3v convertor." Available at https://components101.com/sites/default/files/component_datasheet/FT232RL-USB-TO-TTL-Converter-Datasheet.pdf (09/06/2024).

[22] Robu.in, "5-36v switch drive high-power mosfet trigger module." Available at https://robu.in/product/switch-drive-high-power-mosfet-trigger-module/ (09/06/2024).

[23] Ato, "Rotary encoder with push switch." Available at https://www.ato.com/12v-dc-8-5a-100w-switching-power-supply (10/06/2024).

[24] TRINAMIC Motion Control GmbH Co., "Tmc2209 datasheet." Available at https://www.analog.com/TMC2209/datasheet (09/06/2024).

[25] K. Magdy, "Stm32 hal library tutorial." Available at https://deepbluembedded.com/stm32-hal-library-tutorial-examples/ (10/06/2024).

[26] S. Hymel, "Getting started with stm32 - how to use spi." Available at https://www.digikey.com/en/maker/projects/getting-started-with-stm32-how-to-use-spi/09eab3dfe74c4d0391aaaa99b0a8ee17 (09/06/2024).

[27] STMicroelectronics, "Universal asynchronous serial communications." Available at https://community.st.com/ysqtg83639/attachments/ysqtg83639/stm32-mcu-products-forum/61266/1/USART.pdf (09/06/2024).

[28] R. S. Khandpur, *Printed Circuit Boards: Design, Fabrication, and Assembly.* New York: McGraw Hill, 2005.

[29] Neotel technology, "What is pick and place machine?." Available at https://global.neotel.tech/2023/01/12/what-is-pick-and-place-machine/ (08/06/2024).

[30] Yamaha, "Ultra-high-speed modular z:ta-r ysm40r overview." Available at https://global.yamaha-motor.com/business/smt/mounter/ysm40r/ (08/06/2024).

[31] Advanced assembly, "The pick and place machine unveiled." Available at https://aapcb.com/new-blog/the-pick-and-place-machine-unveiled/ (05/06/2024).

[32] CNC Machines, "What are cnc machines?." Available at https://cncmachines.com/what-is-a-cnc-machine (06/06/2024).

[33] J. Kuusama, "liteplacer." Available at https://liteplacer.com/ (09/06/2024).

[34] Microsmt, "A01-microsmt pnpv3 machine- for openpnp." Available at https://www.microsmt.com.cn/products/microsmt-pnpmachine-v3-for-openpnp (09/06/2024).

[35] ProtoExpress, "Different smd component package sizes." Available at https://www.protoexpress.com/kb/different-smd-component-package-sizes/ (13/06/2024).

[36] Wikipedia, "Ball screw." Available at https://en.wikipedia.org/wiki/Ball$_s crew$(06/06/2024).

[37] RBS, "Ball screw vs lead screw: Everything you need to know." Available at https://rockfordballscrew.com/ball-screw-vs-lead-screw-everything-you-need-to-know-2/ (07/06/2024).

[38] C. Layosa, "Strengths limitations: Belt drive vs. ball screw actuators." Available at https://us.misumi-ec.com/blog/strengths-limitations-belt-drive-vs-ball-screw-actuators/ (07/06/2024).

[39] Isotech, "Belt-driven versus ball screw actuator: Which is the best choice for your application?." Available at https://www.isotechinc.com/belt-driven-versus-ball-screw-actuators/ (07/06/2024).

[40] linuxCNC, "Best wiring practices." Available at https://linuxcnc.org docs/2.8/html/integrator/wiring.html (09/06/2024).

[41] Power electric, "Speed vs torque." Available at https://www.powerelectric.com/motor-blog/speed-vs-torque (01/06/2024).

[42] portescap, "Precision accuracy." Available at https://www.portescap.com/en/solutions/motor-precision-and-accuracy (01/06/2024).

[43] Oriental motor, "Speed — torque curves for stepper motors." Available at https://www.orientalmotor.com/stepper-motors/technology/speed-torque-curves-for-stepper-motors.html (02/06/2024).

[44] STEPPER ONLINE, "Brushless dc motors vs. stepper motors." Available at https://www.omc-stepperonline.com/support/brushless-dc-motors-vs-stepper-motors (02/06/2024).

[45] J. F. Young, *ELEC 201 Course notes and resources*. Rice University, Houston, Texas: Online publication, 2000.

[46] Thomasnet, "Stepper motors vs. dc motors – what's the difference?." Available at https://www.thomasnet.com/articles/machinery-tools-supplies/stepper-motors-vs-dc-motors/ (02/06/2024).

[47] Wikipedia, “Stepper motor.” Available at https://en.wikipedia.org/wiki/Stepper_motor (03/06/2024).

[48] Monolithic power systems, “Stepper motors basics: Types, uses, and working principles.” Available at https://www.monolithicpower.com/stepper-motors-basics-types-uses (29/05/2024).

[49] Portescap, “A guide to stepper motor terminology and parameters.” Available at https://www.portescap.com/en/newsroom/whitepapers/2023/08/a-guide-to-stepper-motor-terminology-and-parameters (29/05/2024).

[50] D. Collins, “Detent torque and holding torque.” Available at https://www.motioncontroltips.com/faq-whats-the-difference-between-detent-torque-and-holding-torque/ (29/05/2024).

[51] ROBOTIK SISTEM, “Stepper motor type properties.” Available at https://www.robotiksistem.com/stepper_motor_types_properties.html (25/05/2024).

[52] Oriental motor, “Basics of stepper motors.” Available at https://www.orientalmotor.com/stepper-motors/technology/stepper-motor-basics.html (25/05/2024).

[53] D. Collins, “How does the number of stator phases affect stepper motor performance?.” Available at https://www.linearmotiontips.com/how-does-the-number-of-stator-phases-affect-stepper-motor-performance/ (13/06/2024).

[54] NEMA, “Motors and generators.” Available at https://www.nema.org/standards/view/motors-and-generators (09/06/2024).

[55] Botland, “Stepper motor jk57hs76-2804 200 steps/rot 3v / 2,8a / 1,89nm.” Available at https://botland.store/stepper-motors/14553-stepper-motor-jk57hs76-2804-200-stepsrot-3v-28a-189nm-5904422342524.html (09/06/2024).

[56] HTA3D, “Nema 17 stepper motor - 17hs8401 - 42hs48 - 42-48 - 5mm d shaft.” Available at https://www.hta3d.com/en/nema-17-stepper-motor-17hs8401-42hs48-42-48-5mm-d-shaft (10/06/2024).

[57] Botnroll, “Nema 17 stepper motor for 3d printer - 42bygh48-23d.” Available at https://www.botnroll.com/en/stepper-motor/1563-nema-17-stepper-motor-with-connector-and-wire-1-m-for-reprap-3d-printer.html (09/06/2024).

[58] Nanotec, “Sca2818l1504-l – stepper motor with hollow shaft – nema 11.” Available at https://www.nanotec.com/eu/en/products/1269-hollow-shaft-motors (10/06/2024).

[59] IQSdirectory, “Vacuum pumps.” Available at https://www.iqsdirectory.com/articles/vacuum-pump.html (09/06/2024).

[60] A. Yard, “Vacuum pressure.” Available at https://groups.google.com/ g/openpnp/c/4RhhiEOYvMA/m/_tBxby8WAwAJ (09/06/2024).

[61] All3DP, “Stepper motor driver: All you need to know.” Available at https://all3dp.com/2/what-s-a-stepper-motor-driver-why-do-i-need-it/ (09/06/2024).

[62] P. Millet, "Why microstepping in stepper motors isn't as good as you think." Available at https://www.ednasia.com/why-microstepping-in-stepper-motors-isnt-as-good-as-you-think/ (09/06/2024).

[63] Faulhaber, "Stepper motor technical note: Microstepping myths and realities." Available at https://www.faulhaber.com/en/know-how/tutorials/stepper-motor-tutorial-microstepping-myths-and-realities/ (09/06/2024).

[64] Texas Instruments, "Drv8825 stepper motor controller ic." Available at https://www.ti.com/lit/gpn/DRV8825 (09/06/2024).

[65] TOSHIBA, "Tb6600hg." Available at https://www.mouser.com/ds/2/408/TB6600HG-483084.pdf (09/06/2024).

[66] D. Collins, "What is a constant voltage drive for a stepper motor and when is it used?." Available at https://www.motioncontroltips.com/what-is-constant-voltage-drive-for-stepper-motor-and-when-is-it-used/ (09/06/2024).

[67] D. Collins, "Stepper drives: What's the difference between an l/r drive and a chopper drive?." Available at https://www.motioncontroltips.com/stepper-drives-whats-the-difference-between-an-l-r-drive-and-a-chopper-drive/ (09/06/2024).

[68] B. Lutkevich, "microcontroller (mcu)." Available at https://www.techtarget.com/iotagenda/definition/microcontroller/ (09/06/2024).

[69] Protoneer, "Arduino cnc shield – 100% grbl compatable." Available at https://blog.protoneer.co.nz/arduino-cnc-shield/License (09/06/2024).

[70] Scienci Labs, "Limit switches." Available at https://resources.sienci.com/view/lmk2-limit-switches/ (09/06/2024).

[71] B. Wood, "Home and limit switches." Available at https://www.bobscnc.com/blogs/more-about-cnc/home-and-limit-switches (09/06/2024).

[72] Open Impulse, "1.8" spi lcd module (128×160)." Available at https://www.openimpulse.com/blog/products-page/product-category/1-8-spi-lcd-module-128x160/ (09/06/2024).

[73] Dejan, "How rotary encoder works and how to use it with arduino." Available at https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/ (09/06/2024).

[74] Matrix Orbital, "Communication protocol." Available at https://www.matrixorbital.com/communication-protocol/ (09/06/2024).

[75] Fernhill SCADA, "Serial communications." Available at https://www.fernhillsoftware.com/help/ drivers/ serial-communication/index.html/ (09/06/2024).

[76] Monolithic Power Systems, "Understanding ac/dc power supplies." Available at https://www.monolithicpower.com/en/ac-dc-power-supply-basics (09/06/2024).

[77] R. Hat, "What is an ide?." Available at https://www.redhat.com/en/topics/middleware/what-is-ide (09/06/2024).

[78] STMicroelectronics, "Stm32cubeide user guide." Available at https://www.st.com/resource/en/user_manual/dm00629856-description-of-the-integrated-development-environment-for-stm32-products-stmicroelectronics.pdf (09/06/2024).

[79] STMicroelectronics, "Description of stm32f4 hal and low-layer drivers." Available at https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf (09/06/2024).

[80] L. Petersen, "Part 1: Introduction to the stm32 microcontroller clock system." Available at https://community.st.com/t5/stm32-mcus/part-1-introduction-to-the-stm32-microcontroller-clock-system/ta-p/605369 (09/06/2024).

[81] M. Harris, "How important is your microcontroller clock source?." Available at https://resources.altium.com/p/how-important-your-microcontroller-clock-source-0 (09/06/2024).

[82] S. Hymel, "Getting started with stm32 - timers and timer interrupts." Available at https://www.digikey.com/en/maker/projects/getting-started-with-stm32-timers-and-timer-interrupts/d08e6493cefa486fb1e79c43c0b08cc6 (09/06/2024).

[83] S. Shahryiar, "Stm32 timers." Available at https://embedded-lab.com/blog/stm32-timers/ (09/06/2024).

[84] STM32 Wiki, "Getting started with gpio." Available at https://wiki.st.com/stm32mcu/wiki/Getting_started_with_GPIO#What_is_a_general_purpose_input_output_-GPIO- (09/06/2024).

[85] Cadence, "Comparing uart vs. spi speed." Available at https://resources.pcb.cadence.com/blog/2022-comparing-uart-vs-spi-speed (09/06/2024).

[86] L. Carolo, "3d printer g-code commands: Main list quick tutorial." Available at https://all3dp.com/2/3d-printer-g-code-commands-list-tutorial/ (09/06/2024).

[87] S. S. Skogsrud, "Grbl wiki." Available at https://github.com/gnea/grbl/wiki (09/06/2024).

[88] Afiskon, "stm32-st7735." Available at https://github.com/afiskon/stm32-st7735/tree/master (09/06/2024).

[89] veysiadn, "tmc_2209." Available at https://github.com/veysiadn/tmc_2209/tree/main (09/06/2024).

[90] J. I. Quinones, "Applying acceleration and deceleration profiles to bipolar stepper motors," tech. rep.

[91] Michael Stumpf, Dmitry Xmelkov, "strtod.c." Available at https://onlinedocs.microchip.com/pr/GUID-317042D4-BCCE-4065-BB05-AC4312DBC2C4-en-US-2/index.html?GUID-2FFC928A-236E-4AAB-A8E1-BCE0B91498E1 (09/06/2024).

Appendix A

Anex

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

//#include "st7735.h"
#include "stdlib.h"
#include "fonts.h"
#include "string.h"
#include "math.h"
#include "stdio.h"
#include "tmc2209.h"
#include "tmc2209_defines.h"
```

```
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define LEFT 0
#define RIGHT 1
#define UP 0
#define DOWN 1
#define FORWARDS 1
#define BACKWARDS 0

#define TEST 0
#define WORK 1

#define MILLIMETRES 0
#define MIL 1 // One mil = 0.02 millimeters, when this option is employed, the
    X-axis stepper motor driver should be set to 1/16th microstepping

#define INTERNAL_PWM_FREQUENCY_23KHZ 0 // Actual frequency is 23.44 kHz
#define INTERNAL_PWM_FREQUENCY_35KHZ 1 // Actual frequency is 35.15 kHz
#define INTERNAL_PWM_FREQUENCY_46KHZ 2 // Actual frequency is 46.51 kHz
#define INTERNAL_PWM_FREQUENCY_58KHZ 3 // Actual frequency is 58.82 kHz
#define MICROSTEPS_PER_STEP_X 1
#define MICROSTEPS_PER_STEP_Y 1
#define MICROSTEPS_PER_STEP_Z 1
#define MICROSTEPS_PER_STEP_A 1
#define RUN_CURRENT_PERCENT  70

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

//Position struct
struct coordinates {
```

```
  uint32_t XPosMilli;
  uint32_t YPosMilli;
  uint32_t ZPosMilli;
  uint32_t XPosMil;
  uint32_t YPosMil;
  uint32_t ZPosMil;
  uint32_t XPosSteps;
  uint32_t YPosSteps;
  uint32_t ZPosSteps;
  bool ARotation;
};

//Global variables
struct coordinates position;
uint32_t count;
uint8_t homeScreenOption;
uint8_t testScreenOption;
uint8_t selectedOperationUnit = MILLIMETRES;

uint8_t homeFlag;
uint8_t endFlagTim1;
uint8_t endFlagTim2;
uint8_t endFlagTim3;
uint8_t endFlagTim13;
uint32_t pulses;
uint32_t counterTim1;
uint32_t counterTim2;
uint32_t counterTim3;
uint32_t counterTim13;
uint8_t endOfInput;

//Message buffers
char RxData[1];
char RxBuffer[1000];

//Stepper drivers
tmc2209_stepper_driver_t stepperDriverY;
tmc2209_stepper_driver_t stepperDriverZ;
tmc2209_stepper_driver_t stepperDriverA;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */

//general functions

void work(void);
```

```
int main(void);

//visual functions
void showHomeScreen(void);
void showTestScreen(void);
uint8_t interactHomeScreen(void);
uint8_t interactTestScreen(void);

//move functions
void moveStepperMotorX(uint32_t steps, uint8_t direct);
void moveStepperMotorY(uint32_t steps, uint8_t direct);
void moveStepperMotorZ(uint32_t steps, uint8_t direct);
void rotateStepperMotorA(uint32_t degrees);
void sendStepsX(uint32_t steps, uint8_t direction);
void sendStepsZ(uint32_t steps, uint8_t direction);
void sendStepsY(uint32_t steps, uint8_t direction);
void sendStepsA(uint32_t steps, uint8_t direction);
void pickUpComponent(void);
void putDownComponent(void);
void employNozzle(void);
void depositNozzle(void);
void home(void);

//calibration function
void getZMotorMaxSteps(void);

//debug functions
void printUsart2(char tx_data[]);
void putCharUsart2(char ch2[1]);

//error handlers
void errorHandler2(void);

//help functions
uint32_t millimeterStepConversion(double millimeters, uint8_t motor);
uint32_t milStepConversion(double mil, uint8_t motor);
uint8_t readNumber(char charBuffer[], uint8_t charCounter, float *floatPtr);
void setupDrivers(void);
uint8_t pushButton(void);

//component test functions
//void testEndstops(void);
//void testMotorsShield(void);
//void pickUpComponentReel1(void);
//void pickUpComponentReel2(void);
//void manualTest();

/* USER CODE END PFP */
```

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_TIM2_Init();
  MX_TIM3_Init();
  MX_USART3_UART_Init();
  MX_UART5_Init();
  MX_TIM13_Init();
  MX_TIM5_Init();
  /* USER CODE BEGIN 2 */

  //Enable the stepper motors on the CNC shield
  HAL_GPIO_WritePin(SHIELD_ENABLE_GPIO_Port, SHIELD_ENABLE_Pin, GPIO_PIN_RESET);

  //Start the potentiometer button timers so it can act as encoder
  HAL_TIM_Encoder_Start(&htim5, TIM_CHANNEL_ALL);
  //Defines whether a homing sequence is expected, if its is not and endstop is hit
     -> malfunction
```

```
homeFlag = 0;
//Driver setup
setupDrivers();

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */

while (1){

   home();
   homeScreenOption = interactHomeScreen();
   switch (homeScreenOption){

      case TEST:
       HAL_Delay(200);

         __HAL_TIM_SET_AUTORELOAD(&htim5, 3);
         testScreenOption = interactTestScreen();
         switch(testScreenOption){

            //X-motor
               case 0:
                moveStepperMotorX(100, LEFT);
                HAL_Delay(100);
                moveStepperMotorX(100, RIGHT);
            break;

            //Y-motor
            case 1:
                  moveStepperMotorY(100, FORWARDS);
                  HAL_Delay(100);
                  moveStepperMotorY(100, BACKWARDS);
            break;

            //Z-motor
            case 2:
                  moveStepperMotorZ(100, UP);
                  HAL_Delay(100);
                  moveStepperMotorZ(100, DOWN);
            break;

            //A-motor
            case 3:
                turnComponent();
            break;
         }
```

```
        break;

        case WORK:

           // Receive and execute Gcodes over USART
           work();
        break;
     }


    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 200;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
```

```
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */

//Stops the motor when the desired position is reached

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
   if(htim->Instance == TIM1){
       counterTim1++;
      if(counterTim1 >= pulses){
         HAL_TIM_PWM_Stop_IT(&htim1, TIM_CHANNEL_3);
         HAL_TIM_Base_Stop_IT(&htim1);
         endFlagTim1 = 1;
      }
   }
   if (htim->Instance == TIM2){
       counterTim2++;
      if(counterTim2 >= pulses){
         HAL_TIM_PWM_Stop_IT(&htim2, TIM_CHANNEL_2);
         HAL_TIM_Base_Stop_IT(&htim2);
         endFlagTim2 = 1;
      }
   }
   if (htim->Instance == TIM3){
       counterTim3++;
      if(counterTim3 >= pulses){
         HAL_TIM_PWM_Stop_IT(&htim3, TIM_CHANNEL_2);
         HAL_TIM_Base_Stop_IT(&htim3);
         endFlagTim3 = 1;
      }
   }
   if (htim->Instance == TIM13){
       counterTim13++;
      if(counterTim13 >= pulses){
         HAL_TIM_PWM_Stop_IT(&htim13, TIM_CHANNEL_1);
         HAL_TIM_Base_Stop_IT(&htim13);
         endFlagTim13 = 1;
```

```
      }
   }
}

// Moves to the nozzle holder and employs the currently deposited nozzle

void employNozzle(){
   home();

   HAL_Delay(1000);

   moveStepperMotorX(105,RIGHT);
   moveStepperMotorZ(4750,DOWN);

   HAL_Delay(1000);

   moveStepperMotorX(5, RIGHT);
   moveStepperMotorZ(10, UP);
   moveStepperMotorX(5, RIGHT);
   moveStepperMotorZ(10, UP);
   moveStepperMotorX(5, RIGHT);
   moveStepperMotorZ(10, UP);
   moveStepperMotorX(5, RIGHT);
   moveStepperMotorZ(10, UP);
   moveStepperMotorX(400, RIGHT);

   home();

}

// Moves to the nozzle holder and deposits the currently employed nozzle in the
    holder

void depositNozzle(){
     home();

     HAL_Delay(1000);
     moveStepperMotorX(450,RIGHT); //with 1/4 microstepping enabled
     moveStepperMotorZ(4500 ,DOWN);
     HAL_Delay(1000);
     moveStepperMotorX(375, LEFT);
     HAL_Delay(1000);
     moveStepperMotorZ(500,UP);

     home();

}

/*
```

```
// Moves to the first reel and picks up a component

void pickUpComponentReel1(){
     home();
     HAL_Delay(1000);
     moveStepperMotorX(975,RIGHT); //with 1/4 microstepping enabled
     moveStepperMotorZ(4650,DOWN);
     pickUpComponent();
     HAL_Delay(2000);
}

// Moves to the second reel and picks up a component

void pickUpComponentReel2(){
     home();
     HAL_Delay(1000);
     moveStepperMotorX(1215,RIGHT);//with 1/4th microstepping enabled
     moveStepperMotorZ(4635,DOWN);
     pickUpComponent();
     HAL_Delay(2000);
}

// Function to control the motor manually when four buttons are installed

void manualTest(){

   uint16_t xsteps = 0;
   uint16_t zsteps = 0;
   char debug[1000];

   __HAL_TIM_SET_AUTORELOAD(&htim1, 699);
   __HAL_TIM_SET_AUTORELOAD(&htim2, 399);
   __HAL_TIM_SET_AUTORELOAD(&htim3, 299);
   __HAL_TIM_SET_AUTORELOAD(&htim13, 299);

   while(1){
      if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_14) == 0){
         sendStepsZ(5, DOWN);
         zsteps+=5;
         sprintf(debug, "total zsteps = %u", zsteps);
         printUsart2(debug);
      }
      else if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_15) == 0){
         sendStepsZ(5, UP);
         zsteps-=5;
         sprintf(debug, "total zsteps = %u", zsteps);
         printUsart2(debug);
      }
      else if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_1) == 0){
```

```
            sendStepsX(5, RIGHT);
            xsteps+=5;
            sprintf(debug, "total xsteps = %u", xsteps);
            printUsart2(debug);
        }
        else if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_2) == 0){
            sendStepsX(5, LEFT);
            xsteps-=5;
            sprintf(debug, "total xsteps = %u", xsteps);
            printUsart2(debug);
        }
    }
}
*/

// Converts millimeters into step, X = 1, Y = 2, Z = 3

uint32_t millimeterStepConversion(double millimeters, uint8_t motor){

    uint32_t steps;

    if (motor == 1){
        steps = round(millimeters * 3.3333 * MICROSTEPS_PER_STEP_X);
    }
    else if (motor == 2){
        steps = round(millimeters * 250 * MICROSTEPS_PER_STEP_Y);
    }
    else if (motor == 3){
        steps = round(millimeters * 50 * MICROSTEPS_PER_STEP_Z);
    }

    return steps;

}

// Converts mil into step, X = 1, Y = 2, Z = 3

uint32_t milStepConversion(double mil, uint8_t motor){

    uint32_t steps;

    if (motor == 1){
        if (MICROSTEPS_PER_STEP_X >= 16){
            steps = mil * MICROSTEPS_PER_STEP_X;
        }
        else{
            printUsart2("Operation in this mode is only accurate when 1/16 microstepping
                or higher is selected for the X-axis \n");
            printUsart2("Exiting program \n");
```

```
        exit(0);

      }
   }
   else if (motor == 2){
      steps = mil * 5 * MICROSTEPS_PER_STEP_Y;
   }
   else if (motor == 3){
      steps = mil * MICROSTEPS_PER_STEP_Z;
   }

   return steps;
}

// Home the motors to their starting positions, this function should be executed
// before any other to calibrate the positions of the motor

void home(){

     printUsart2(" --- Homing motors to starting position --- \n");
     homeFlag = 1;

     __HAL_TIM_SET_AUTORELOAD(&htim1, 699);
     __HAL_TIM_SET_AUTORELOAD(&htim2, 399);
     __HAL_TIM_SET_AUTORELOAD(&htim3, 299);
     __HAL_TIM_SET_AUTORELOAD(&htim13, 299);

     sendStepsZ(20000, UP);
     sendStepsY(20000, BACKWARDS);
     sendStepsX(20000, LEFT);


     homeFlag = 0;
     position.XPosMilli = 0;
     position.YPosMilli = 0;
     position.ZPosMilli = 0;
     position.XPosMil = 0;
     position.YPosMil = 0;
     position.ZPosMil = 0;
     position.XPosSteps = 0;
     position.YPosSteps = 0;
     position.ZPosSteps = 0;
}

//Initialize and setup all TMC2209 drivers on the CNC shield

void setupDrivers(){

   uint8_t buffer_vis[80];
```

```
printUsart2(" Setup A axis stepper driver \n");
tmc2209_setup(&stepperDriverA, 115200, SERIAL_ADDRESS_3);
tmc2209_set_hardware_enable_pin(&stepperDriverA, GPIO_PIN_9);
enable_cool_step(&stepperDriverA, 1, 0);
tmc2209_enable(&stepperDriverA);
set_micro_steps_per_step(&stepperDriverA, MICROSTEPS_PER_STEP_A);
set_pwm_frequency(&stepperDriverA, INTERNAL_PWM_FREQUENCY_46KHZ);
set_stand_still_mode(&stepperDriverA, TMC_NORMAL);
set_all_current_percent_values(&stepperDriverA, RUN_CURRENT_PERCENT, 0, 0);
enable_automatic_current_scaling(&stepperDriverA);
enable_stealth_chop(&stepperDriverA);
set_stealth_chop_duration_threshold(&stepperDriverA, 9999999);

HAL_Delay(100);

printUsart2(" Setup Y axis stepper driver \n");
tmc2209_setup(&stepperDriverY, 115200, SERIAL_ADDRESS_1);
tmc2209_set_hardware_enable_pin(&stepperDriverY, GPIO_PIN_9);
enable_cool_step(&stepperDriverY, 1, 0);
tmc2209_enable(&stepperDriverY);
set_micro_steps_per_step(&stepperDriverY, MICROSTEPS_PER_STEP_Y);
set_pwm_frequency(&stepperDriverY, INTERNAL_PWM_FREQUENCY_46KHZ);
set_stand_still_mode(&stepperDriverY, TMC_NORMAL);
set_all_current_percent_values(&stepperDriverY, RUN_CURRENT_PERCENT, 0, 0);
enable_automatic_current_scaling(&stepperDriverY);
enable_stealth_chop(&stepperDriverY);
set_stealth_chop_duration_threshold(&stepperDriverY, 9999999);

HAL_Delay(100);

printUsart2(" Setup Z axis stepper driver \n\n");
tmc2209_setup(&stepperDriverZ, 115200, SERIAL_ADDRESS_2);
tmc2209_set_hardware_enable_pin(&stepperDriverZ, GPIO_PIN_9);
enable_cool_step(&stepperDriverZ, 1, 0);
tmc2209_enable(&stepperDriverZ);
set_micro_steps_per_step(&stepperDriverZ, MICROSTEPS_PER_STEP_Z);
set_pwm_frequency(&stepperDriverZ, INTERNAL_PWM_FREQUENCY_46KHZ);
set_stand_still_mode(&stepperDriverZ, TMC_NORMAL);
set_all_current_percent_values(&stepperDriverZ, RUN_CURRENT_PERCENT, 0, 0);
enable_automatic_current_scaling(&stepperDriverZ);
enable_stealth_chop(&stepperDriverZ);
set_stealth_chop_duration_threshold(&stepperDriverZ, 9999999);

HAL_Delay(100);

sprintf(buffer_vis, " Stepper driver Y with address: %.1u is running in 1/%.1u
    microsteps \n",stepperDriverY.serial_address_,
    get_microstep_per_step(&stepperDriverY));
```

```
    printUsart2(buffer_vis);
    sprintf(buffer_vis, " Stepper driver Z with address: %.1u is running in 1/%.1u
        microsteps \n",stepperDriverZ.serial_address_,
        get_microstep_per_step(&stepperDriverZ));
    printUsart2(buffer_vis);
    sprintf(buffer_vis, " Stepper driver A with address: %.1u is running in 1/%.1u
        microsteps \n",stepperDriverA.serial_address_,
        get_microstep_per_step(&stepperDriverA));
    printUsart2(buffer_vis);

    printUsart2("\n --- END TMC2209 Configuration --- \n");
}

//Handles the interaction with the homescreen

uint8_t interactHomeScreen(void){

    char selectedOption[2];
    uint16_t valuePot, prevValuePot;

     valuePot = 2;
    showHomeScreen();
    while (pushButton() == 1 ){

        prevValuePot = valuePot;
        valuePot = __HAL_TIM_GetCounter(&htim5);
        sprintf(selectedOption, "%.1u", valuePot);
        ST7735_WriteString(1, 72, selectedOption, Font_11x18, ST7735_GREEN,
            ST7735_WHITE);
        if ( valuePot != prevValuePot ){

           switch (valuePot){

              case TEST: // Test Motor with potentiometer - 0
                  ST7735_FillRectangle(10, 62, 110, 34, ST7735_BLUE );
                  ST7735_WriteString(40, 70, "TEST", Font_11x18, ST7735_BLACK,
                      ST7735_BLUE);
                  ST7735_FillRectangle(10, 102, 110, 34, ST7735_YELLOW );
                  ST7735_WriteString(40, 110, "WORK", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
              break;

              case WORK: // Do some work with Gcodes - 1
                  ST7735_FillRectangle(10, 62, 110, 34, ST7735_YELLOW );
                  ST7735_WriteString(40, 70, "TEST", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
                  ST7735_FillRectangle(10, 102, 110, 34, ST7735_BLUE );
                  ST7735_WriteString(40, 110, "WORK", Font_11x18, ST7735_BLACK,
                      ST7735_BLUE);
```

```
            break;
          }
        }
        HAL_Delay(200);
    }
    return valuePot;
}

//Handles the interaction with the test screen

uint8_t interactTestScreen(void){

    char selectedOption[2];
    uint16_t valuePot, prevValuePot;

     valuePot = 2;
     showTestScreen();
    while ( pushButton() == 1 )
    {
        prevValuePot = valuePot;
        valuePot = __HAL_TIM_GetCounter(&htim5);
        sprintf(selectedOption, "%.1u", valuePot);
        ST7735_WriteString(1, 72, selectedOption, Font_11x18, ST7735_GREEN,
            ST7735_WHITE);
        if ( valuePot != prevValuePot ){
           switch (valuePot)
           {
              case 0:
                  ST7735_FillRectangle(10, 20, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 20, "X", Font_11x18, ST7735_BLACK,
                      ST7735_BLUE);
                  ST7735_FillRectangle(10, 50, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 50, "Y", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
                  ST7735_FillRectangle(10, 80, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 80, "Z", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
                  ST7735_FillRectangle(10, 110, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 110, "A", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
              break;

              case 1:
                  ST7735_FillRectangle(10, 20, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 20, "X", Font_11x18, ST7735_BLACK,
                      ST7735_YELLOW);
                  ST7735_FillRectangle(10, 50, 110, 20, ST7735_YELLOW );
                  ST7735_WriteString(60, 50, "Y", Font_11x18, ST7735_BLACK,
                      ST7735_BLUE);
```

```
                ST7735_FillRectangle(10, 80, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 80, "Z", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 110, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 110, "A", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
            break;

            case 2:
                ST7735_FillRectangle(10, 20, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 20, "X", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 50, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 50, "Y", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 80, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 80, "Z", Font_11x18, ST7735_BLACK,
                    ST7735_BLUE);
                ST7735_FillRectangle(10, 110, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 110, "A", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
            break;

            case 3:
                ST7735_FillRectangle(10, 20, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 20, "X", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 50, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 50, "Y", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 80, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 80, "Z", Font_11x18, ST7735_BLACK,
                    ST7735_YELLOW);
                ST7735_FillRectangle(10, 110, 110, 20, ST7735_YELLOW );
                ST7735_WriteString(60, 110, "A", Font_11x18, ST7735_BLACK,
                    ST7735_BLUE);
                //HAL_Delay(500);
            break;
          }
        }
        HAL_Delay(200);
    }
    return valuePot;
}

//Controls the movement of the X-axis motor and controls the speed following an
    acceleration profile

void moveStepperMotorX(uint32_t steps, uint8_t direct){
```

```
uint32_t longMovementARRX[] = {800, 750, 725, 700, 675, 650, 635};
uint32_t shortMovementARRX[] = {800, 750, 800};

uint32_t pulses = 0;
 uint32_t pos = 0;
 uint16_t numberOfRisingSteps = sizeof(longMovementARRX) /
    sizeof(longMovementARRX[0]);;
 float risePercentage = 0.2;


if(steps >= 400){

   // Rising speed portion of the movement
   for (int i = 0; i < numberOfRisingSteps; i++){

      __HAL_TIM_SET_AUTORELOAD(&htim1, longMovementARRX[i]);
      sendStepsX(steps * risePercentage / numberOfRisingSteps, direct);
      pos += steps * risePercentage / numberOfRisingSteps;
   }

   // Maximum speed portion of the movement
   pulses = steps * (1-(2*risePercentage));
   __HAL_TIM_SET_AUTORELOAD(&htim1, longMovementARRX[numberOfRisingSteps-1]);
   sendStepsX(pulses, direct);
   pos += pulses;

   // Falling speed portion of the movement
   for (int i = numberOfRisingSteps; i > 0; i--){

      __HAL_TIM_SET_AUTORELOAD(&htim1, longMovementARRX[i-1]);
      sendStepsX(steps * risePercentage / numberOfRisingSteps, direct);
      pos += steps * risePercentage / numberOfRisingSteps;
   }

   // Correction for rounding errors during calculation
   sendStepsX(steps-pos, direct);
   pos += steps - pos;
}

// Short movement so only need 3 levels
else{

   pulses = steps / 3;
   __HAL_TIM_SET_AUTORELOAD(&htim1, shortMovementARRX[0]);
   sendStepsX(pulses, direct);
   pos += pulses;

   __HAL_TIM_SET_AUTORELOAD(&htim1, shortMovementARRX[1]);
```

```
      sendStepsX(pulses, direct);
      pos += pulses;

      pulses = steps - 2 * pulses;
      __HAL_TIM_SET_AUTORELOAD(&htim1, shortMovementARRX[2]);
      sendStepsX(pulses, direct);
      pos += pulses;
   }

   // Update global position of head
   if (direct == RIGHT){
      position.XPosSteps += pos;
   }
   else{
      position.XPosSteps -= pos;
   }

   HAL_Delay(10);
}

//Controls the movement of the Y-axis motor and controls the speed following an
    acceleration profile

void moveStepperMotorY(uint32_t steps, uint8_t direct){

   uint32_t longMovementARRY[] = {499, 399, 299, 199, 149, 129, 119};
   uint32_t shortMovementARRY[] = {499,299,499};

   uint32_t pulses = 0;
    uint32_t pos = 0;
    uint16_t numberOfRisingSteps = sizeof(longMovementARRY) /
        sizeof(longMovementARRY[0]);;
    float risePercentage = 0.2;

   if(steps >= 1000){

      // Rising speed portion of movement
      for (int i = 0; i < numberOfRisingSteps; i++){

         __HAL_TIM_SET_AUTORELOAD(&htim2, longMovementARRY[i]);
         sendStepsY(steps * risePercentage / numberOfRisingSteps, direct);
         pos += steps * risePercentage / numberOfRisingSteps;

      }

      // Maximum speed portion of movement
      pulses = steps * (1-(2*risePercentage));
      __HAL_TIM_SET_AUTORELOAD(&htim2, longMovementARRY[numberOfRisingSteps-1]);
      sendStepsY(pulses, direct);
```

```
        pos += pulses;

        // Falling speed portion of movement
        for (int i = numberOfRisingSteps; i > 0; i--){

            __HAL_TIM_SET_AUTORELOAD(&htim2, longMovementARRY[i-1]);
            sendStepsY(steps * risePercentage / numberOfRisingSteps, direct);
            pos += steps * risePercentage / numberOfRisingSteps;
        }

        // Correction for rounding errors during calculation
        sendStepsY(steps-pos, direct);
        pos += steps - pos;
    }

    // Short movement so only need 3 levels
    else{

        pulses = steps / 3;
        __HAL_TIM_SET_AUTORELOAD(&htim2, shortMovementARRY[0]);
        sendStepsY(pulses, direct);
        pos += pulses;

        __HAL_TIM_SET_AUTORELOAD(&htim2, shortMovementARRY[1]);
        sendStepsY(pulses, direct);
        pos += pulses;

        pulses = steps - 2 * pulses;
        __HAL_TIM_SET_AUTORELOAD(&htim2, shortMovementARRY[2]);
        sendStepsY(pulses, direct);
        pos += pulses;
    }

    // Update global position of head
    if (direct == FORWARDS){
        position.YPosSteps += pos;
    }
    else{
        position.YPosSteps -= pos;
    }
    HAL_Delay(10);
}

//Controls the movement of the Z-axis motor and controls the speed following an
    acceleration profile

void moveStepperMotorZ(uint32_t steps, uint8_t direct){
```

```
//uint32_t longMovementARRZ[] = {1999, 1499, 999, 799, 549, 499, 459, 399, 379,
    349};
uint32_t longMovementARRZ[] = {349, 299, 249, 199, 149, 129};
uint32_t shortMovementARRZ[] = {349,199,349};

uint32_t pulses = 0;
 uint32_t pos = 0;
 uint16_t numberOfRisingSteps = sizeof(longMovementARRZ) /
     sizeof(longMovementARRZ[0]);;
 float risePercentage = 0.2;

if(steps >= 1000){

   //Rising speed portion of the movement
   for (int i = 0; i < numberOfRisingSteps; i++){

      __HAL_TIM_SET_AUTORELOAD(&htim3, longMovementARRZ[i]);
      sendStepsZ(steps * risePercentage / numberOfRisingSteps, direct);
      pos += steps * risePercentage / numberOfRisingSteps;

   }

   // Maximum speed portion of movement
   pulses = steps * (1-(2*risePercentage));
   __HAL_TIM_SET_AUTORELOAD(&htim3, longMovementARRZ[numberOfRisingSteps-1]);
   sendStepsZ(pulses, direct);
   pos += pulses;

   // Falling speed portion of the movement
   for (int i = numberOfRisingSteps; i > 0; i--){

      __HAL_TIM_SET_AUTORELOAD(&htim3, longMovementARRZ[i-1]);
      sendStepsZ(steps * risePercentage / numberOfRisingSteps, direct);
      pos += steps * risePercentage / numberOfRisingSteps;
   }

   // Correction for rounding errors during calculation
   sendStepsZ(steps-pos, direct);
   pos += steps - pos;
}

// Short movement so only need 3 levels
else{

   pulses = steps / 3;
   __HAL_TIM_SET_AUTORELOAD(&htim3, shortMovementARRZ[0]);
   sendStepsZ(pulses, direct);
   pos += pulses;
```

```
      __HAL_TIM_SET_AUTORELOAD(&htim3, shortMovementARRZ[1]);
      sendStepsZ(pulses, direct);
      pos += pulses;

      pulses = steps - 2 * pulses;
      __HAL_TIM_SET_AUTORELOAD(&htim3, shortMovementARRZ[2]);
      sendStepsZ(pulses, direct);
      pos += pulses;
   }

   // Update global position of head
   if (direct == DOWN){
      position.ZPosSteps += pos;
   }
   else{
      position.ZPosSteps -= pos;
   }
   HAL_Delay(10);
}

//Rotate the A-axis over a certain number of degrees

void rotateStepperMotorA(uint32_t degrees){

   if (degrees == 90){
      sendStepsA(50, 0);
      position.ARotation = !position.ARotation;
   }
   else if (degrees == 180){
      sendStepsA(100, 0);
   }
   else{
      sendStepsA(round(degrees*0.55555), 0);
   }
}


//Handles the interaction with the push button

uint8_t pushButton(void)
{
       if(HAL_GPIO_ReadPin(PUSH_BUTTON_GPIO_Port, PUSH_BUTTON_Pin) == 0) //is the
          pushbutton pressed (input = 0)
       {
            HAL_Delay(10);   //wait 10ms for debounce
            if(HAL_GPIO_ReadPin(PUSH_BUTTON_GPIO_Port, PUSH_BUTTON_Pin) == 0) //is
               the pushbutton still pressed?
                  return 0;  //true button push
            else
```

```
                return 1;  //false from contact bounce
        }
        return 1;
}

//Enables solenoid and vacuum pump in order to pick up component

void pickUpComponent(){

   HAL_GPIO_WritePin(VACUUM_PUMP_GPIO_Port, VACUUM_PUMP_Pin, GPIO_PIN_SET);
   HAL_GPIO_WritePin(SOLENOID_GPIO_Port, SOLENOID_Pin, GPIO_PIN_SET);
}

//Disables solenoid and vacuum pump in order to place component

void putDownComponent(){

   HAL_GPIO_WritePin(VACUUM_PUMP_GPIO_Port, VACUUM_PUMP_Pin, GPIO_PIN_RESET);
   HAL_GPIO_WritePin(SOLENOID_GPIO_Port, SOLENOID_Pin, GPIO_PIN_RESET);
}

//Displays the home screen on the ST7735

void showHomeScreen(void){

   ST7735_Init();
   ST7735_Init();

   ST7735_FillScreen(ST7735_WHITE);
   HAL_Delay(10);

   ST7735_WriteString(10, 2, "PNP MACH", Font_11x18, ST7735_RED, ST7735_WHITE);

   ST7735_FillRectangle(10, 62, 110, 34, ST7735_YELLOW );
   ST7735_WriteString(40, 70, "TEST", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
   HAL_Delay(100);

   ST7735_FillRectangle(10, 102, 110, 34, ST7735_YELLOW );
   ST7735_WriteString(40, 110, "WORK", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
   HAL_Delay(100);

   return;
}

//Displays the test screen on the ST7735

void showTestScreen(void){

   ST7735_FillScreen(ST7735_WHITE);
```

```
    HAL_Delay(10);

    ST7735_FillRectangle(10, 20, 110, 20, ST7735_YELLOW );
    ST7735_WriteString(60, 20, "X", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
    HAL_Delay(10);
    ST7735_FillRectangle(10, 50, 110, 20, ST7735_YELLOW );
    ST7735_WriteString(60, 50, "Y", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
    HAL_Delay(10);
    ST7735_FillRectangle(10, 80, 110, 20, ST7735_YELLOW );
    ST7735_WriteString(60, 80, "Z", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
    HAL_Delay(10);
    ST7735_FillRectangle(10, 110, 110, 20, ST7735_YELLOW );
    ST7735_WriteString(60, 110, "A", Font_11x18, ST7735_BLACK, ST7735_YELLOW);
    HAL_Delay(10);

    return;
}

//Send the steps to the X-motor by enabling the PWM timer

void sendStepsX(uint32_t steps, uint8_t direction){

    uint8_t homingDone = 0;
    counterTim1 = 0;
     pulses = steps;
     endFlagTim1 = 0;

    if (direction == RIGHT){
       HAL_GPIO_WritePin(CNC_X_DIR_GPIO_Port, CNC_X_DIR_Pin, GPIO_PIN_SET);
    }
    else {
       HAL_GPIO_WritePin(CNC_X_DIR_GPIO_Port, CNC_X_DIR_Pin, GPIO_PIN_RESET);
    }

    HAL_TIM_Base_Start_IT(&htim1);
    HAL_TIM_PWM_Start_IT(&htim1, TIM_CHANNEL_3);
    while(endFlagTim1 == 0){

        if (!HAL_GPIO_ReadPin(CNC_X_LIMIT_GPIO_Port, CNC_X_LIMIT_Pin)){

             HAL_Delay(10);   //wait 10ms for debounce
             if (!HAL_GPIO_ReadPin(CNC_X_LIMIT_GPIO_Port, CNC_X_LIMIT_Pin)){ //is the
                 end stop still pressed? -> debounce routine

            // if homeFLag is activated and endstop is touched move 50 steps back from
            // endstop so that it is no longer pressed and make this position zero
            if (homeFlag == 1){
               if (homingDone == 0){
                  printUsart2("Homing sequence X-axis complete\n");
```

```
                homingDone = 1;
              }
              counterTim1 = 0;
              pulses = 50;
              HAL_GPIO_WritePin(CNC_X_DIR_GPIO_Port, CNC_X_DIR_Pin, GPIO_PIN_SET);
            }
            else{
              printUsart2("End stop unexpectedly pressed -> exiting operation");
              exit(0);
            }
             }
          else{
              //False press from noise
             }
       }

   };
   HAL_Delay(1);
}

//Send the steps to the Y-motor by enabling the PWM timer

void sendStepsY(uint32_t steps, uint8_t direction){

   uint8_t homingDone = 0;
   counterTim2 = 0;
    pulses = steps;
    endFlagTim2 = 0;

   if (direction == BACKWARDS){
      HAL_GPIO_WritePin(CNC_Y_DIR_GPIO_Port, CNC_Y_DIR_Pin, GPIO_PIN_SET);
   }
   else {
      HAL_GPIO_WritePin(CNC_Y_DIR_GPIO_Port, CNC_Y_DIR_Pin, GPIO_PIN_RESET);
   }

   HAL_TIM_Base_Start_IT(&htim2);
   HAL_TIM_PWM_Start_IT(&htim2, TIM_CHANNEL_2);
   while(endFlagTim2 == 0){

       if (!HAL_GPIO_ReadPin(CNC_Y_LIMIT_GPIO_Port, CNC_Y_LIMIT_Pin)){
         HAL_Delay(10);
         if (!HAL_GPIO_ReadPin(CNC_Y_LIMIT_GPIO_Port, CNC_Y_LIMIT_Pin)){ //is the end
             stop still pressed? -> debounce routine

            // if homeFLag is activated and endstop is touched move 50 steps back from
            // endstop so that it is no longer pressed and make this position zero
            if (homeFlag == 1){
              if (homingDone == 0){
```

```
                printUsart2("Homing sequence Y-axis complete\n");
                homingDone = 1;
            }
            counterTim2 = 0;
            pulses = 50;
            HAL_GPIO_WritePin(CNC_Y_DIR_GPIO_Port, CNC_Y_DIR_Pin, GPIO_PIN_RESET);
         }
         else{
            printUsart2("End stop unexpectedly pressed -> exiting operation");
            exit(0);
         }
      }
      else{
         //False press from noise
      }
    }
  };
  HAL_Delay(1);
}

//Send the steps to the Z-motor by enabling the PWM timer

void sendStepsZ(uint32_t steps, uint8_t direction){

  uint8_t homingDone = 0;
  counterTim3 = 0;
   pulses = steps;
   endFlagTim3 = 0;

  if (direction == 0){
    HAL_GPIO_WritePin(CNC_Z_DIR_GPIO_Port, CNC_Z_DIR_Pin, GPIO_PIN_SET);
  }
  else {
    HAL_GPIO_WritePin(CNC_Z_DIR_GPIO_Port, CNC_Z_DIR_Pin, GPIO_PIN_RESET);
  }

  HAL_TIM_Base_Start_IT(&htim3);
  HAL_TIM_PWM_Start_IT(&htim3, TIM_CHANNEL_2);
  while(endFlagTim3 == 0){

      if (!HAL_GPIO_ReadPin(CNC_Z_LIMIT_GPIO_Port, CNC_Z_LIMIT_Pin)){
        HAL_Delay(10);
        if (!HAL_GPIO_ReadPin(CNC_Z_LIMIT_GPIO_Port, CNC_Z_LIMIT_Pin)){
          // if homeFLag is activated and endstop is touched move 50 steps back from
          // endstop so that it is no longer pressed and make this position zero
          if (homeFlag == 1){
            if (homingDone == 0){
              printUsart2("Homing sequence Z-axis complete\n");
              homingDone = 1;
```

```
            }
            counterTim3 = 0;
            pulses = 50;
            HAL_GPIO_WritePin(CNC_Z_DIR_GPIO_Port, CNC_Z_DIR_Pin, GPIO_PIN_RESET);
          }
          else{
            printUsart2("End stop unexpectedly pressed -> exiting operation");
            exit(0);
          }
        }
        else{
          //False press from noise
        }
      }
  };
  HAL_Delay(1);
}

//Send the steps to the A-motor by enabling the PWM timer

void sendStepsA(uint32_t steps, uint8_t direction){

  counterTim13 = 0;
   pulses = steps;
   endFlagTim13 = 0;

  if (direction == 0){
    HAL_GPIO_WritePin(CNC_A_DIR_GPIO_Port, CNC_A_DIR_Pin, GPIO_PIN_SET);
  }
  else {
    HAL_GPIO_WritePin(CNC_A_DIR_GPIO_Port, CNC_A_DIR_Pin, GPIO_PIN_RESET);
  }

  HAL_TIM_Base_Start_IT(&htim13);
  HAL_TIM_PWM_Start_IT(&htim13, TIM_CHANNEL_1);
  while(endFlagTim13 == 0){};
  HAL_Delay(1);
}


// Functions that takes G-codes as input and controls the machine

void work(void){

  printUsart2(" --- Entering work mode --- \n");

  uint8_t charCounter, index, numberLength;
  char letter, axis, debugBuff[1000], test[1000];;
  float value;
```

```
uint32_t intValue, zSpeed;
int32_t zMovement, xMovement, yMovement;
bool end;

// initialize values
endOfInput = 0;
charCounter = 0;
index = 0;
end = false;

// USART1 has baudrate of 9600, 1 stop bit, no parity bit
ST7735_WriteString(2, 140, "waiting on RX", Font_7x10, ST7735_YELLOW,
    ST7735_BLACK);
printUsart2(" --- Waiting for data --- \n");


while (1)
{
   while (end != true){
      HAL_UART_Receive_IT(&huart3, (uint8_t*) RxData, 1);
      if (*RxData != 0){
         if (*RxData != 47){

            //DEBUG STATEMENT
            //sprintf(test,"| received %d character, character = %c |", index + 1,
                *RxData);
            //ST7735_WriteString(2, 140, "character received", Font_7x10,
                ST7735_YELLOW, ST7735_BLACK);
            //printUsart2(test);

            RxBuffer[index] = *RxData;
            index++;
            *RxData = 0;
         }
         else{
            end = true;
            sprintf(test,"\n Received Gcode string: %s \n\n", RxBuffer);
            printUsart2(test);

         }
      }

   }

   // End of input is indicated by user by entering the '/' symbol
   if (endOfInput == 0)
      {
               intValue = 0;
```

```
// Import the next g-code word, expecting a letter followed by a
    value. Otherwise, error out.
letter = RxBuffer[charCounter];
charCounter ++;

// If first character in the beginning of the input isn't 'G' or 'M'
    invalid Gcode -> error out
if (letter != 'G' && letter != 'M'){
   sprintf(debugBuff, "COMMAND CAN'T START WITH --> %c, EXITING THE
      PROGRAM\n", letter);
   printUsart2(debugBuff);
   exit(0);
}


// Read the number after the letter, if its not a valid number,
    error out
// function returns the total length of the number including the
    leading zeros to keep the char count correct
// returns 0 if there is no number
value = 0;
numberLength = readNumber(RxBuffer, charCounter, &value);
if (numberLength == 0) {
   sprintf(debugBuff, "UNEXPECTED CHARACTER AFTER --> %c, EXITING
      THE PROGRAM\n", letter);
   printUsart2(debugBuff);
   exit(0);
}


// Update the character counter with the length of the number so
    that it is on the last number before the next letter
charCounter = charCounter + numberLength;

// get the length of the number without leading zeros
intValue = truncf(value);

 //DEBUG STATEMENT
 //sprintf(buff_long,"%u", int_value);
 //sprintf(buff_mov,"number contained %u chars, char_counter con
     b_long = %u \n", strlen(buff_long), char_counter);
 //printUsart2(buff_mov);

// Check if the g-code word is supported or errors due to modal
    group violations
switch(letter) {

    // 'G' and 'M' Command Words: Parse commands and check for modal
       group violations.
```

```
case 'G':

    // Determine 'G' command and its modal group
    switch(intValue){

        case 1:

            // Read next letter to determine the axis to move
            axis = RxBuffer[charCounter];

            // Skip all the spaces until axis is found
            while (axis == ' '){
                charCounter++;
                axis = RxBuffer[charCounter];
            }

             //DEBUG STATEMENT
             //sprintf(buff_mov,"axis = %c \n", axis);
             //printUsart2(buff_mov);


            if (axis == 'X'){

                charCounter++;

                //No number or invalid number after axis -> error
                    out
                numberLength = readNumber(RxBuffer, charCounter,
                    &value);
                if (numberLength == 0) {
                 printUsart2("STATUS-1_BAD_NUMBER_FORMAT, EXITING
                     THE PROGRAM \n");
                 exit(0);
                }

                //get the length of the number to move the char
                    counter
                intValue = truncf(value);

                ST7735_WriteString(80, 140, "Moving X-axis",
                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
            sprintf(debugBuff, " Moving X-axis to position: %.1u
                \n", intValue);
            printUsart2(debugBuff);

            if (selectedOperationUnit == MILLIMETRES){
               xMovement = intValue - position.XPosMilli;
                    if (xMovement > 0){
```

```
            moveStepperMotorX(millimeterStepConversion(abs(xMovement)
                1), RIGHT);
            }
            else{
            moveStepperMotorX(millimeterStepConversion(abs(xMovement)
                1), LEFT);
            }
            position.XPosMilli += xMovement;
            position.XPosMil += xMovement * 50;
    }

    else{
       xMovement = intValue - position.XPosMil;
            if (xMovement > 0){
            moveStepperMotorX(milStepConversion(abs(xMovement),
                1), RIGHT);
            }
            else{
            moveStepperMotorX(milStepConversion(abs(xMovement),
                1), LEFT);
            }
            position.XPosMil += xMovement;
            position.XPosMilli = round(position.XPosMil *
               50);
    }

        charCounter += numberLength;

        // if the character behind an axis is a space ->
            expect another axis to be defined
        // otherwise, this command has ended and move on
            to the next
        if (RxBuffer[charCounter] == ' '){
         charCounter++;
         axis = RxBuffer[charCounter];
        }
        else{
         charCounter++;
         break;
        }
     }

     if (axis == 'Y'){

        charCounter++;

        //No number or invalid number after axis -> error
            out
```

```
    numberLength = readNumber(RxBuffer, charCounter,
        &value);
    if (numberLength == 0) {
     printUsart2("STATUS-1_BAD_NUMBER_FORMAT, EXITING
         THE PROGRAM \n");
     exit(0);
    }

    //get the length of the number to move the char
        counter
    intValue = truncf(value);

    ST7735_WriteString(80, 140, "Moving Y-axis",
        Font_7x10, ST7735_YELLOW, ST7735_BLACK);
sprintf(debugBuff, " Moving Y-axis to position: %.1u
    \n", intValue);
printUsart2(debugBuff);

if (selectedOperationUnit == MILLIMETRES){
   yMovement = intValue - position.YPosMilli;
        if (yMovement > 0){
         moveStepperMotorY(millimeterStepConversion(abs(yMovement)
             1), RIGHT);
        }
        else{
         moveStepperMotorY(millimeterStepConversion(abs(yMovement)
             1), LEFT);
        }
        position.YPosMilli += yMovement;
        position.YPosMil += yMovement * 50;
}

else{
   yMovement = intValue - position.YPosMil;
        if (yMovement > 0){
         moveStepperMotorY(milStepConversion(abs(yMovement),
             1), RIGHT);
        }
        else{
         moveStepperMotorY(milStepConversion(abs(yMovement),
             1), LEFT);
        }
        position.YPosMil += yMovement;
        position.YPosMilli = round(position.YPosMil *
            50);
}

    charCounter += numberLength;
```

```
        // if the character behind an axis is a space ->
            expect another axis to be defined
        // otherwise, this command has ended and move on
            to the next
        if (RxBuffer[charCounter] == ' '){
         charCounter++;
         axis = RxBuffer[charCounter];

        }
        else{
         charCounter++;
         break;
        }
    }

    if (axis == 'Z'){

        zSpeed = intValue;
        charCounter++;

        //No number or invalid number after axis -> error
            out
        numberLength = readNumber(RxBuffer, charCounter,
            &value);
        if (numberLength == 0) {
         printUsart2("STATUS-2_BAD_NUMBER_FORMAT, EXITING
             THE PROGRAM \n");
         exit(0);
        }

        //get the length of the number to move the char
            counter
        intValue = truncf(value);

        ST7735_WriteString(80, 140, "Moving Z-axis",
            Font_7x10, ST7735_YELLOW, ST7735_BLACK);
    sprintf(debugBuff, " Moving Z-axis to position: %.1u
        \n", intValue);
    printUsart2(debugBuff);

    if (selectedOperationUnit == MILLIMETRES){
       zMovement = intValue - position.ZPosMilli;
            if (zMovement > 0){
             moveStepperMotorZ(millimeterStepConversion(abs(zMovement)
                 1), RIGHT);
            }
            else{
             moveStepperMotorZ(millimeterStepConversion(abs(zMovement)
                 1), LEFT);
```

```
        }
        position.ZPosMilli += zMovement;
        position.ZPosMil += zMovement * 50;
}

else{
   zMovement = intValue - position.ZPosMil;
        if (zMovement > 0){
         moveStepperMotorZ(milStepConversion(abs(zMovement),
             1), RIGHT);
        }
        else{
         moveStepperMotorZ(milStepConversion(abs(zMovement),
             1), LEFT);
        }
        position.ZPosMil += zMovement;
        position.ZPosMilli = round(position.ZPosMil *
            50);
}
     charCounter += numberLength;

     // if the character behind an axis is a space ->
         expect another axis to be defined
     // otherwise, this command has ended and move on
         to the next
     if (RxBuffer[charCounter] == ' '){
      charCounter++;
      axis = RxBuffer[charCounter];
     }
     else{
      charCounter++;
      break;
     }
 }

 if (axis == 'A'){

     charCounter++;

     //No number or invalid number after axis -> error
         out
     numberLength = readNumber(RxBuffer, charCounter,
         &value);
     if (numberLength == 0) {
      printUsart2("STATUS-2_BAD_NUMBER_FORMAT, EXITING
          THE PROGRAM \n");
      exit(0);
     }
```

```
            //get the length of the number to move the char
                counter
            intValue = truncf(value);

            ST7735_WriteString(80, 140, "Rotating A-axis",
                Font_7x10, ST7735_YELLOW, ST7735_BLACK);
        sprintf(debugBuff, " Rotating A over %.1u degrees \n",
            intValue);
        printUsart2(debugBuff);


            rotateStepperMotorA(intValue);

            charCounter += numberLength;

            // if the character behind an axis is a space ->
                expect another axis to be defined
            // otherwise, this command has ended and move on
                to the next
            if (RxBuffer[charCounter] == ' '){
             charCounter++;
             axis = RxBuffer[charCounter];
            }
            else{
             charCounter++;
             break;
            }
         }

         break;

      case 4:

char waitTime = RxBuffer[charCounter];

// Skip all the spaces until time is found
while (waitTime == ' '){
   charCounter++;
   waitTime = RxBuffer[charCounter];
}

numberLength = readNumber(RxBuffer, charCounter, &value);
if (numberLength == 0) {
   printUsart2("STATUS-1_BAD_NUMBER_FORMAT, EXITING THE
       PROGRAM \n");
   exit(0);
}
//get the length of the number to move the char counter
intValue = truncf(value);
```

```
            sprintf(debugBuff, " Wating %.1u milliseconds \n",
                intValue);
            printUsart2(debugBuff);

         HAL_Delay(intValue);
         charCounter += numberLength + 1;

         break;

            case 20:
               charCounter++;
                printUsart2(" Use mil as unit \n");
                selectedOperationUnit = MIL;
                ST7735_WriteString(80, 140, "Using mil as unit",
                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                HAL_Delay(1000);
                break;

            case 21:
               charCounter++;
                printUsart2(" Use millimeters as unit \n");
                selectedOperationUnit = MILLIMETRES;
                ST7735_WriteString(80, 140, "Using millimeter as
                    unit", Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                HAL_Delay(1000);
                break;

            case 28: // Homes the motors
               charCounter++;
               printUsart2(" Homing the motors \n");
               home();
               ST7735_WriteString(80, 140, "homing the motors",
                   Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                break;

            default: printUsart2("STATUS_GCODE_UNSUPPORTED_COMMAND
                --> EXITING THE PROGRAM \n"); // [Unsupported G
                command]
      }
      break;

   case 'M':
      //Determine 'M' command and its modal group
      switch(intValue){
         case 3:
               charCounter++;
                 printUsart2(" Picking up component \n");
```

```
                                ST7735_WriteString(80, 140, "Pick up component",
                                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                                pickUpComponent();
                            break;
                            case 5:
                                charCounter++;
                                printUsart2(" Releasing component \n");
                                ST7735_WriteString(80, 140, "Release component",
                                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                                putDownComponent();
                            break;
                            case 6:
                                charCounter++;
                                printUsart2(" employing toolhead \n");
                                ST7735_WriteString(80, 140, "employ toolhead",
                                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                                employNozzle();
                            break;
                            case 7:
                                charCounter++;
                                printUsart2(" depositing toolhead \n");
                                ST7735_WriteString(80, 140, "deposit toolhead",
                                    Font_7x10, ST7735_YELLOW, ST7735_BLACK);
                                depositNozzle();
                            break;
                            default: printUsart2("STATUS_MCODE_UNSUPPORTED_COMMAND
                                \n"); // [Unsupported M command]
                        }


                }

                 //DEBUG STATEMENT

                 //sprintf(buff_mov,"| index = %u, char counter = %u |", index,
                     char_counter);
                 //printUsart2(buff_mov);


                if (charCounter + 1 >= index){
                    endOfInput = 1;
                    printUsart2(" \n End of program \n");
                }

            }
    }
}

//Print a char buffer over USART2
```

```
void printUsart2(char text[])
{
     if(HAL_UART_Transmit(&huart2, (uint8_t*)text, (uint16_t) strlen(text), 1000) !=
         HAL_OK)
     {
        errorHandler2();
     }
     return;
}

//Print a single char over USART2

void putCharUsart2(char ch2[1])
{
   HAL_UART_Transmit(&huart2, (uint8_t*) ch2, 1, 1000);
}

/**
 *           Extracts a floating point value from a string. The following code is
    based loosely on
 *         the avr-libc strtod() function by Michael Stumpf and Dmitry Xmelkov and
    many freely
 *           available conversion method examples, but has been highly optimized for
    Grbl. For known
 *         CNC applications, the typical decimal value is expected to be in the
    range of E0 to E-4.
 *         Scientific notation is officially not supported by g-code, and the 'E'
    character may
 *         be a g-code word on some CNC systems. So, 'E' notation will not be
    recognized.
 *         NOTE: Thanks to Radu-Eosif Mihailescu for identifying the issues with
    using strtod().
  */

uint8_t readNumber(char charBuffer[], uint8_t charCounter, float *floatPtr)
{
    uint8_t charIndex, valueIndex;
    uint8_t c;
    uint32_t valor;
   char valueBuff[12];
   float fval;

   valueIndex = 0;
   memset(valueBuff, '?', sizeof valueBuff);
   charIndex = charCounter;
    c = charBuffer[charIndex];

   while (c >= '0' && c <= '9'){
```

```
      valueBuff[valueIndex] = c;
      valueIndex++;
      charIndex++;
      c = charBuffer[charIndex];
   }
   if (strlen((char*)valueBuff) == 0) return(0);
   charCounter = charIndex;
   valor = atoi((char*)valueBuff);
    fval = (float)valor;
    *floatPtr = fval;

   return(valueIndex);
}


void errorHandler2(void)
{
  while(1)
  {
   printUsart2("Error");
   HAL_Delay (200);
  }
}



/* USER CODE END 4 */

/**
  * @brief This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT
/**
  * @brief Reports the name of the source file and the source line number
  *        where the assert_param error has occurred.
  * @param file: pointer to the source file name
```

```
  * @param line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```