FMH606 Master's Thesis 2022
Electrical Power Engineering

Development of an open control interface for a servo machine test stand

Anniken Semb Kvalsund

Faculty of Technology, Natural Sciences and Maritime Sciences
Campus Porsgrunn

University of
South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis 2022
Title: *Development of an open control interface for a servo machine test stand*
Pages: *125*
Keywords: *servo machine, load simulation, open control, Python, Arduino*

Student: *Anniken Semb Kvalsund*
Supervisor: *Dietmar Winkler*
External partner: *University of South-Eastern Norway*

Summary:
A few years ago, the University of South-Eastern Norway acquired a Servo Machine Test stand, consisting of an asynchronous servo machine with a corresponding drive coupled to a small frequency drive controlled asynchronous machine. The stand is used for small-scale load-handling demonstrations and was planned to be used in teaching and research settings. The servo machine is configured as a brake, mimicking various load conditions, controlled by either a physical user panel or a computer program via a USB interface. However, as the supplied software only included a narrow range of applications, the goal is to develop a more flexible control interface allowing for further simulations and control applications.

The thesis analyses the test stand's components and their restrictions, including its communication options, including CANopen, LenzeDiag and analogue I/O terminals, and considers the most viable one to be the analogue I/O terminals due to the serial ports secured access and cost.

A new control interface is developed based on Python's open-source programming software and Arduino's open-source and accessible hardware. The new interface communicates with the test stand through its I/O terminals via developed electronic amplifiers and creates a solid base for further development towards more extensive hardware in the loop simulations.

Preface

This master's thesis constitutes the final part of the two-year master's program in Electrical Power Engineering at the University of South-Eastern Norway (USN). The thesis was conducted from January to June during the spring semester of 2022 and extends 30 ECTS.

I would like to thank Dietmar Winkler, my supervisor at USN, for providing outstanding support and guidance whenever needed. He lowered the infamous threshold for questions I feared were too dumb to ask and have been a relieving consistent and reliable source of advice after two years filled with COVID uncertainty. I also want to thank Bjørn Vegard Tveraaen and Kjetil Svendsen for their help with microcontrollers and electronics.

Finally, I would like to thank all my classmates who, somehow, have made the two years filled with online and hybrid lectures into an enjoyable time spent in good company.

Porsgrunn, 18th May 2022

Anniken Semb Kvalsund

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

ACK	Acknowledge
$ADDR$	Address
CAN	Controller Area Network
CRC	Cyclic Redundancy Check
CS	Control Signal
CS_{LRV}	Control Signal Lower Range Value
CS_{res}	Controller Signal Resolution
CS_{URV}	Control Signal Upper Range Value
DAQ	Data Acquisition
DIP	Dual In-line Package
EMF	Electromotive Force
$ENDP$	Endpoint
EOF	End of Frame
EOP	End of Package
I/O	Input/Output
IC	Integrated Circuit
ID	Identifier
LRV	Lower Range Value

Max_{output}	Maximum output value
MV	Measured Value
MV_{scaled}	Normalised measured value
$P.ID$	Package Identifier
PF	Power Factor
PLC	Programmable Logic Controller
PWM	Pulse width modulation
RMF	Rotating magnetic field
RTR	Remote Transmission Request
SOF	Start of Frame
SP_{received}	Received setpoint
$SYNCH$	Synchronise
URV	Upper Range Value
USB	Universal Serial Bus
ASCII	American Standard Code for Information Interchange

Variables and constants

N_n	Nominal speed	$[rpm]$
$\dot{m}$	Flow	$[\frac{m^3}{s}]$
ω	Mechanical speed	$[rpm]$
τ	Torque	$[Nm]$
A	Ampere	$[A]$
D	Duty Cycle	$[\%]$
E_2	Induced EMF at standstill	$[V]$

E_2r	Rotor induced EMF	$[V]$
f	Frequency	$[Hz]$
f_r	Rotor frequency	$[Hz]$
l_c	Load constant	
M_{max}	Maximum torque	$[Nm]$
N	Machine speed	$[rpm]$
N_r	Rotational speed	$[rpm]$
N_s	Synchronous speed	$[rpm]$
p	Number of poles	
t_{cycle}	Cycle time	$[s]$
t_{off}	Off-time	$[s]$
t_{on}	On-time	$[s]$
V	Voltage	$[V]$
V_{in}	Input voltage	$[V]$
V_{out}	Output voltage	$[V]$
W	Watt	$[W]$
P	Power	$[W]$
s	Slip	$[\%]$

1 Introduction

As the world slowly turns its back on fossil fuels due to skyrocketing prices and the undeniably negative environmental impact, the widely discussed and inarguable praised hero called electricity is head-on full speed into a new energy supply revolution. The century-old traffic supplying fossil energy to households, factories, automotive, and offshore industries experience occurring changes that dramatically alter the fundamental structure of the trade, leading to a neverending quest for the next, new, best tradeoff. Of course, electricity itself is no new invention in terms of the rapid-evolving community seen over the last six decades. Still, in the vicinity of only a couple of centuries, it has evolved from a compelling, unexplored scientific discovery to one of, if not *the* very staple of the modern world. [1][2]

In the later years, sustainability, green energy and climate consciousness have been the topics on everyone's lips, more or less regardless of one's industry, occupation and geographical identity. A prominent effect of the green energy shift is the extended use of electricity in fields fossil energy sources traditionally dominate, especially in the power generation and the automotive industry. Moreover, the increased use of large electrical machines and the constant strive for energy optimisation results in a pang of hunger for knowledge about the machines' load handling, control systems and power losses, to mention a few. [3][4]

In 2012, the University of South-Eastern Norway acquired a Servo Machine Test stand intended for load-handling demonstrations and other research purposes [5][6]. The stand consists of a servo motor and drive coupled to a small frequency drive controlled asynchronous machine. The frequency drive is controlled through a user panel, while the servo drive either by a physical control panel built around the drive or via a USB interface and its program *ActiveServo*. Unfortunately, the program is somewhat restricted concerning how the different available parameters can be controlled and allow for only a few different load settings. The goal is to develop a more open, adjustable user interface that fits various project needs and allows for more flexible speed and torque control. [6]

In this project, the servo drive test stand's components and capabilities are analysed, along with the software *ActiveServo* and its restrictions. This paper goes through various communication options, their possibilities and limitations and elaborates on the currently most feasible open-source communication method, a microcontroller-based I/O module run by a Python script.

2 Concept

The servo-machine test stand is a complete test system for electrical machines and drives. The stand consists of a digital controller with corresponding software controlling a brake coupled to an induction motor controlled by a variable frequency drive, as illustrated in Figure 2.1

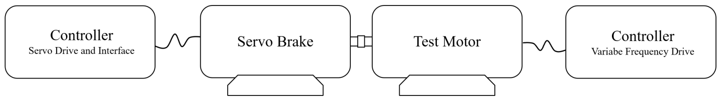


Figure 2.1: Test stand

The test motor runs at a fixed speed controlled by the variable frequency drive (VFD), while the asynchronous servo-brake applies a programmable, variable torque using regenerative braking. With caution not to overload the test motor, the brake can also run to create the effect of a generator.

The servo drive interface stand can either be controlled through its physical user panel, or be connected to and controlled by a computer through a USB interface and the software ActiveServo, a program for examining the responses of load machines [7]. For further details revolving around the user panel and computer program, see subsection 3.2.1 and subsection 3.2.2.

2.1 Induction motors

The system consists of two induction motors, often referenced as asynchronous motors. These are among the most popular motor choices due to their simple but rugged construction, low maintenance, and low price compared to the alternatives [8]. There are two main kinds of three-phase induction motors, the squirrel-cage and the slip ring, the squirrel cage often being the favourite one due to its rugged construction [9].The squirrel cage induction motor consists of a stator with a three-phase winding circuit placed in highly permeable steel laminations and a rotor made of a laminated core with parallel slots carrying conductors aesthetically resembling a squirrel cage [10]. A simplified version without rotor windings is shown in Figure 2.2.

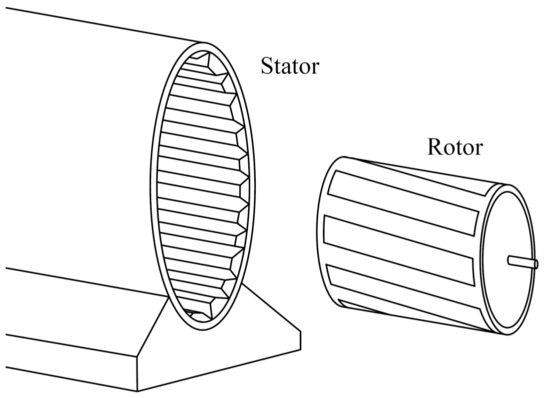


Figure 2.2: Simplified squirrel cage induction motor

The rotor conduction bars are usually not placed parallel to the shaft, but a bit skewed and short-circuited through an end ring to form a complete closed circuit. The slight angle of the bars allows for a more uniform torque curve through different rotor positions, reduces the locking tendency and increases the rotor resistance due to the increased length of the rotor bar conductors [11]. The increased resistance provides a higher starting torque and lowers starting current, the drawback being a bit higher losses during regular operation [12].

The stator consists of a 3-phase winding with metal housing and a core. The windings are placed electrically and mechanically 120° apart, providing a rotating magnetic field when a power source is connected [13]. The windings' terminals are connected in either star or delta in the machine's terminal box. Figure 2.3 illustrates a simple 2-pole, three-phase, star-wound stator wiring diagram.

A three-phase alternating current passes through the windings and produces a rotating magnetic field. ($\boldsymbol{RMS}$). The peed of the machine's internal rotating magnetic field, known as the synchronous speed (N_s), is determined by the number of poles ($\boldsymbol{P}$) and applied frequency (f). The relationship between the motor speed and its pole number is given in Equation (2.1).

$$N_s \propto \frac{1}{p} \quad or \quad N_s = \frac{120f}{p} \tag{2.1}$$

According to Faraday's law, if a closed-loop conductor is placed inside the rotating magnetic field, an electromotive force (EMF), will be induced in the loop, causing the loop, or the rotor, to rotate [10]. A significant advantage of this, especially in squirrel cage motors, is that the motor is inherently self-starting, eliminating the need for external prime movers or damper windings needed in synchronous motors, thus reducing complexity and cost [14].

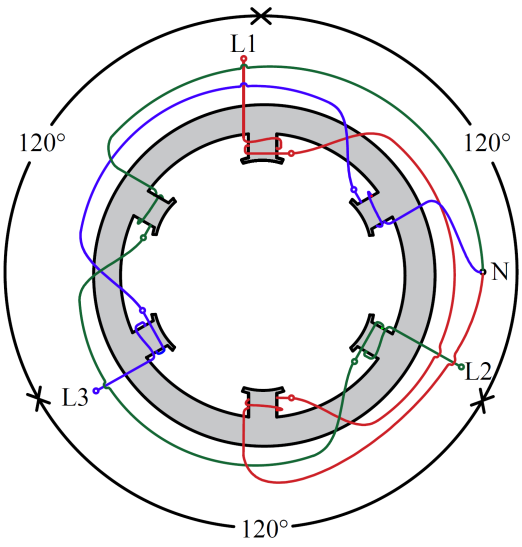


Figure 2.3: Stator windings illustration

As the rotation of the rotor merely relies on the induced EMF, its rotational speed will never be able to catch up to the synchronous speed of the magnetic field, hence the term asynchronous machine. As a result, the rotational speed N_r is always slightly less than the synchronous speed, with the relative difference known as *slip*. The slip is calculated using the synchronous speed N_s and the corresponding rotational speed N_r, as shown in Equation (2.2) [15].

$$s = \frac{N_s - N_r}{N_s} \tag{2.2}$$

The slip can be used to calculate several properties of the motor. Equation (2.3) and Equation (2.4) shows how the rotor frequency f_r, and rotor induced EMF E_2r are calculated. E_2 represents the rotor-induced EMF per phase at a standstill [15].

$$f_r = sf \tag{2.3}$$

$$E_2r = sE_2 \tag{2.4}$$

As described in Section 3.1, the test stand revolves around two motors coupled to each other, one acting as a motor and the other as a regenerative brake. By applying a low-frequency power supply to the brake's stator, causing an RMF-speed lower than the rotor speed, the brake, an asynchronous machine, turns into an inductive generator feeding power back into the supply line [16]. During this process, the slip, as shown in

Equation (2.2), becomes negative, causing a negative torque seen from the brake's point of view. The torque, τ, can be found using the power, P and mechanical speed ω, as described in Equation (2.5) [17]

$$\tau = \frac{P}{\omega} \tag{2.5}$$

Expanding Equation (2.5), the correlation between an inductive machine's torque and slip becomes clear in the induction motor torque equation, presented in Equation (2.6), where I_2 is the rotor current, R_2 the rotor resistance, s the slip and ω_s the synchronous mechanical speed [17].

$$\tau = \frac{3I_2^2R_2}{s\omega_s} \tag{2.6}$$

When using induction motors as regenerative brakes, the stress levels can become very high, especially during quick stops. Therefore, the machine should always be able to handle more power than transferred during the breaking. The brake used in this setup has a nominal power rating equal to more than 4.5 times the motor.

As the rotation of the rotor merely relies on the induced emf, and the number of poles and frequency determines the electromagnetic speed, the common way to control the rotor speed is to alter the frequency by feeding the machine's power supply through a frequency converter.

2.2 Variable Frequency Drive

The two induction motors in the setup are controlled by Variable Frequency Drives (VFDs), which are popular motor control devices that drive electric motors by varying the frequency and voltage of their respective power supplies [18]. As of today, VFDs are found in most industries requiring accurate variable motor speed or torque control, in addition to making their entry into the green technology by improving system efficiencies and reducing mechanical stress on machines during starts and stops by controlling the ramp-up and ramp-downs.

The VFD essentially consists of three main parts: An AC to DC converter, a DC-bus and a DC to AC inverter, as illustrated in Figure 2.4 [19]. The figure pictures a VFD with a three phase input, a single-phased VFD has the an equal structure.

First, the power is fed through the AC to DC converter, often diode, transistors or silicon controlled rectifiers.Diode based rectifiers are the most common ones used due to their low cost and usually consist of a diode bridge paired with a capacitor in parallel to smother

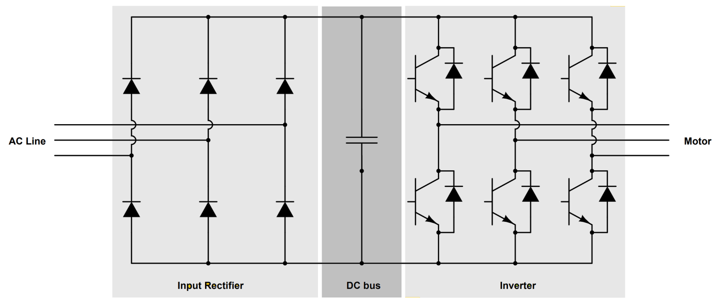
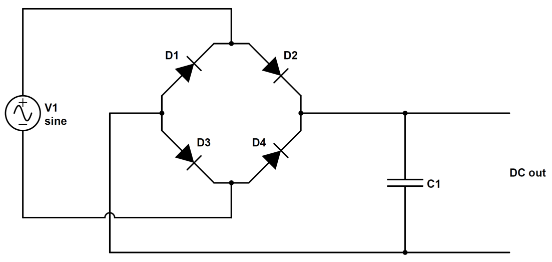
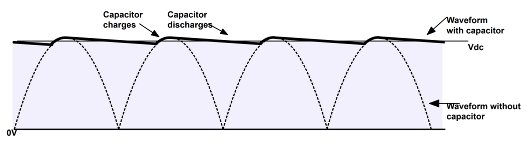


Figure 2.4: Illustration of a frequency drive's three main components. [20]

out the remaining AC ripples [21]. Figure 2.5a pictures a simplified illustration of an ideal diode bridge rectifier and Figure 2.5b what the output waveforms of such a rectifier looks like with and without the added capacitor.



(a) Ideal diode bridge rectifier



(b) Ideal diode bridge rectifier output wave-forms. [21]

Figure 2.5: Ideal diode bridge rectifier

After the rectifier stage, the power runs through the DC bus, consisting of mainly a large capacitor to provide a high-quality low-ripple voltage at the input of the inverter. The DC bus capacitor determines the amount of transient energy the VFD can absorb, and as a consequence, the largest DC bus capacitance within the suitable economic range is often preferred. Therefore, the capacitor is sized not only to minimise the ripple current, but also to provide acceptable compensation for momentary power loss or voltage-sag. [22]

Before the output, the VFD's final stage is the AC inverter stage, where the DC voltage is converted back to AC voltage at desired level and frequency. The currently most widely used technique for voltage inversion is pulse width modulation (***PWM***), controlled by a microcontroller. [23] PWM decides voltage levels with time as its main parameter, switching a voltage on and off on a high frequency, using duty cycles to decide how long the voltage is high or low. A common way to create sinusoidal waves using PWM is to compare the sinusoidal AC voltage with a high-frequency triangular wave in real-time, which determines each switching state for the poles in the inverter, illustrated in Figure 2.6. [24]

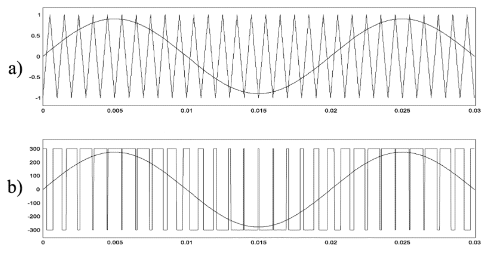
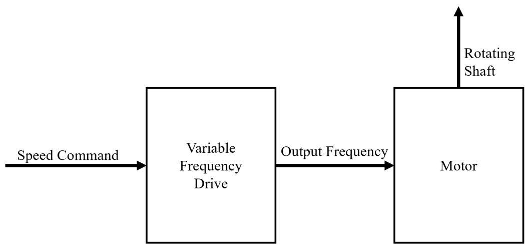


Figure 2.6: PWM generated sinusoidal wave [24]

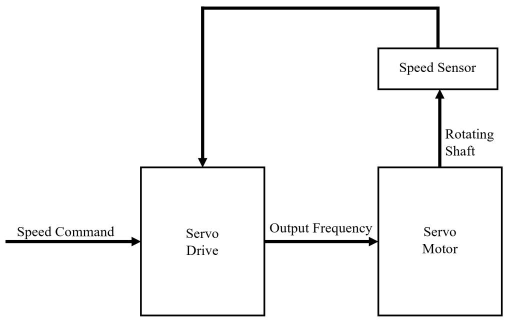
By adjusting the duty cycle and triangular waves, the frequency inverter's output has a wide range and can be used to drive motors efficiently, safely, and accurately.

2.3 Servo Drive

The servo drive controls the servo motor much like the VFD controls the ordinary induction motor, but includes a feedback system that allows for closed-loop control in contrast to the VFD. For example, if one wants to achieve a certain motor speed and adjusts the VFD's output frequency to match the speed, the VFD does not know if the motor reached the desired speed or if, e.g. a connected load caused it to go faster or slower. A feedback loop solves this issue by measuring the relevant value and sending it back to the controller, which in turn adjusts its output value, for example, frequency, accordingly. [25] Figure 2.7 illustrates the an example of the difference between an open-loop (a) control system versus a closed-loop (b) speed control system such as the one used for the servo drive.



(a) Open-loop.



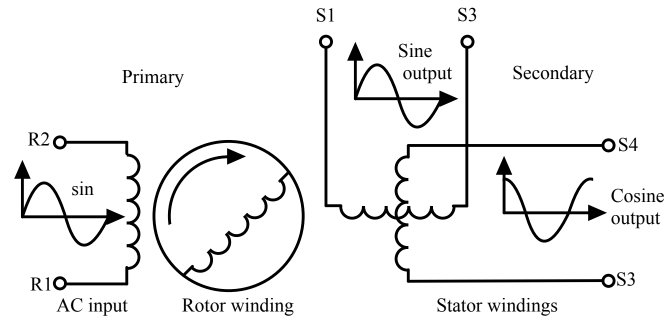
(b) CLosed-loop.

Figure 2.7: Open-loop versus closed-loop control systems

The servo drive used in this project contains a speed sensor in the form of a resolver, a robust sensor consisting of a stationary stator and a rotor attached to the motor shaft.

A resolver can be viewed as a rotating transformer and uses induced voltage to measure speed and angular shaft position.

(a) Resolver example [26].



(b) Resolver windings [27].

Figure 2.8: Resolver

Figure 2.8b illustrates that the stator contains three windings, a primary winding where a fixed induced sine voltage is applied, and two stationary windings mounted 90° apart, named the secondary, or sine and cosine winding. As the shaft rotates, the resolver's alternating magnetic rotor field induces alternating voltages in the sine and cosine windings, where their amplitudes depend on the position of the rotor. The rotor's angular position and speed can then be determined by measuring the induced voltage in the secondary windings.

3 System

3.1 Hardware Description

The test stand, provided by the German technology company Lucas Nülle GmbH[7] consists of an original servo drive controlling a $1.7\,kW$ induction motor acting as a brake to another motor driven by a frequency controller, as shown in Figure 3.1 and Figure 3.1.

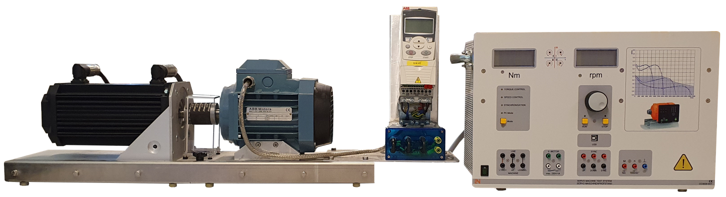


Figure 3.1: Test stand setup.

The servo brake has an internal speed sensor in the form of a resolver connected to the servo drive. The only other feedback sent from the machine rig is a safety sensor checking if the shaft cover is attached correctly (not pictured). If not attached correctly, the drive interface will display an error and shut down the brake.

The power supply from the frequency drive to the motor can optionally be connected through the control panel to allow for some integration, as shown in Figure 3.2. Phase L1 runs through an ammeter and allows the test stand to log the motor's response to the various applied loads.

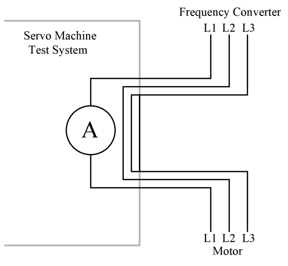


Figure 3.2: Current measurement between frequency converter and motor

Table 3.1: Hardware

Hardware	**Model**
Frequency Converter	ABB ACS360-01E-02A4-2
Test Motor	ABB 3GVA083001-ASC
Servo Drive Test Stand	Lucas Nülle CO3636-6V7
Servo Drive	Lenze E94ASHE0044A22NNNN
Brake Motor	Lenze MCA 13I41-RS0B0-B19N-ST5S00N-ROSU

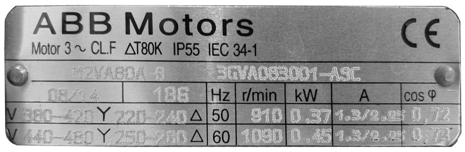
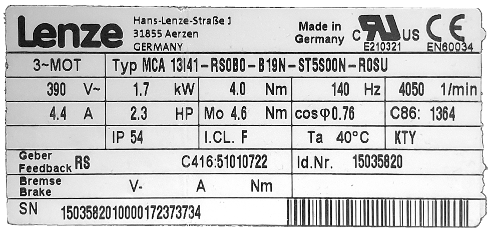


ABB Motors
Motor 3 ~ CL.F ΔT80K IP55 IEC 34-1
M2VA80A-6 3GVA083001-ASC
08/14 186 Hz r/min kW A cos φ
V 380-420 Y 220-240 Δ 50 910 0.37 1.3/2.25 0.72
V 440-480 Y 250-280 Δ 60 1090 0.45 1.3/2.25 0.73

(a) ABB 0.37kW machine nameplate


Lenze
Hans-Lenze-Straße 1
31855 Aerzen
GERMANY
Made in Germany
E210321 EN60034
3~MOT
Typ MCA 13I41-RS0B0-B19N-ST5S00N-ROSU
390 V~ 1.7 kW 4.0 Nm 140 Hz 4050 1/min
4.4 A 2.3 HP Mo 4.6 Nm cosφ 0.76 C86: 1364
IP 54 I.CL. F Ta 40°C KTY
Geber Feedback RS C416:51010722 Id.Nr. 15035820
Bremse Brake V- A Nm
SN 1503582010000172373734

(b) Lenze 1.7kW machine nameplate

Figure 3.3: Machine nameplates

Machines. The test motor is, as presented in Figure 3.3a, a small 0.37kW ABB asynchronous machine, rated $220-240V$ in a delta configuration and $910 rpm$ at $50Hz$, with a power factor PF, often referred to as cos ϕ, of 0.72. The motor and frequency controller differ from the ones initially sold with the test stand, but were chosen in favour of the original ones to reduce costs. In contrast, the brake is a more powerful rated machine selected to withstand the powers caused during regenerative braking. As shown in Figure 3.3b, the brake consists of a 1.7kW Lenze asynchronous machine, rated $390V$ and $050 rpm$ at $140Hz$, with a PF equal to 0.76.

Variable Frequency Drive. The frequency drive ABB ACS350 can be controlled locally, using the control panel mounted on its front, as pictured in Figure 3.4 Using the panel is a relatively quick and user friendly way to operate the drive, albeit it does not allow cascade control or more extensive system integration. Therefore, ABB created multiple additional extension adapters enabling communication through various field bus communication protocols such as TCP/IP, Profibus, CANbus, and others to remote control the frequency drive. The frequency drive in question has a Profibus adapter available. Another control option is to use the provided analogue in- and outputs (I/O) found on the frequency drive underneath the white panel below the local control panel, further described in Section 4.2.

Figure 3.4: ABB ACS350 Variable Frequency Drive

Servo Drive. As described in Table 3.1, the servo drive is a Lenze Highline 9400 Single drive (see Figure 3.5), a three-phased $0.37 - 240kW$ servo inverter with, among tons of other features, a resolver input, a brake resistor to dispose of power generated through braking, digital and analogue I/Os and a variety of optional communication extensions [28]. The drive is for this setup configured as a three-phase $400V$ input at $50Hz$, $400V$ $0 - 140Hz$ output drive using the resolver input as a speed control to match the chosen servo motor. Communication wise, the servo drive has three different options; CANopen, Diagnostics Port and I/O terminals, further described in subsection 3.3.1

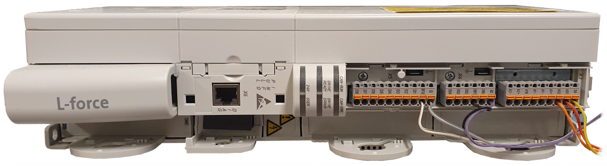


Figure 3.5: Lenze 9400 Highline Servo Drive Front

The servo drive is in the original test stand setup controlled using an interface made by the German technology training company Lucas-Nülle and consists of a physical control panel and a program named ActiveServo, elaborated in Section 3.2.

3.2 Control interface and software

As the VFD covered in Section 2.2 controls the motor, this section (Section 3.2) is dedicated to the servo drive and brake control. The Lucas-Nülle user panel is built for and around the Lenze Servo drive and uses the protocol CANopen, further described in subsubsection 3.3.1, as communication between the drive and the interface. The interface

Table 3.2: Test stand interface features in Figure 3.6.

No.	Feature
1	Torque display
2	Quadrant indicator
3	Speed display
4	Speed / Torque adjuster
5	Mode switch and indicator
6	On / Off switch
7	Motor line in / out for current measurements (Figure 3.2)
8	Thermal sensor and relay output
9	Connection for syncing to grid
10	Torque and speed output in mV
11	USB Port
12	Run / stop buttons

allows for control by using simple commands with buttons and dials on the physical controller or connecting it via a USB to a PC running ActiveServo [29].

3.2.1 User panel

Figure 3.6 shows the front panel of the servo drive controller interface, with its simple layout features described in Table 3.2.

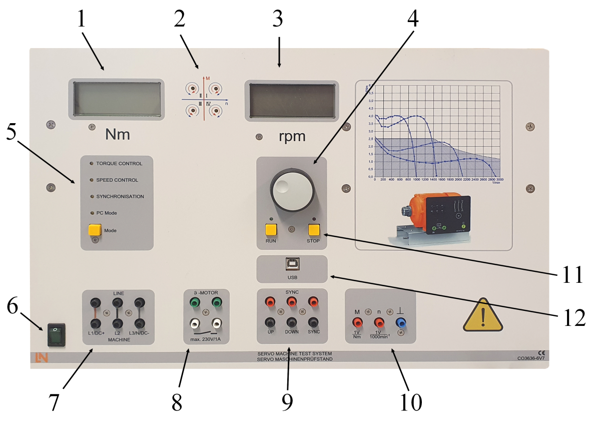


Figure 3.6: Lucas-Nülle servo motor test stand interface

The panel contains two four-digit, seven-segment displays used to show torque and speed or specific errors, such as communication and drive failures. The interface itself cannot record and store data, which means every displayed value is live, and all adjustments are

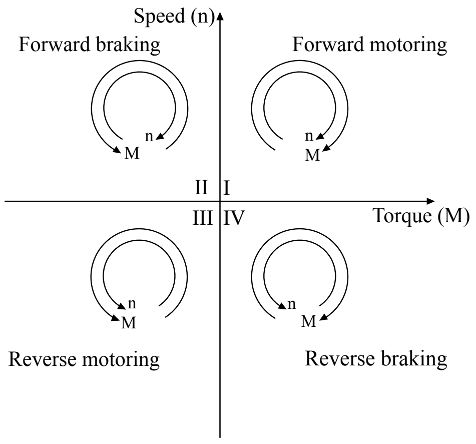


Figure 3.7: Four Quadrant Machine Operation

immediate. Thus, as the servo is operated independently from the test motor, it should be operated with caution to avoid overloading the test motor.

The four-quadrant display (2) shows which quadrant the servo machine operates by switching on or off LEDs in each quadrant. In essence, as illustrated in Figure 3.7, the display indicates in which direction the machine rotates and if it is currently operating as a motor or generator. As the servo machine mainly works as a brake, it will usually operate within the second and fourth quadrant. [28]

3.2.2 ActiveServo

ActiveServo is the program made for comtrolling the Test Stand Setup by Lucas-Nülle and is tailored to examine load machines' static and dynamic responses. The program contains modules to simulate pumps, calendering, hoist machines, inertia wheels and time-dependent loads.

Overview

ActiveServo uses a Windows-based layout with a menu bar containing drop-down menus located at its top, allowing for intuitive navigation between its various applications. For example, Figure 3.8 shows the program's main window, containing a large plotting area with a grid, a machine load parameters tab on the left side of the screen, and a toolbar placed right under the menu bar. The plotting area is the heart of ActiveServo, where the brakes behaviour and the test motor's response are shown during tests. The default units of the x and y-axis are speed [rpm] and torque [Nm], but can be changed according to the user's need and preference. The machine area on the left hand contains real-time displays of the brake's recorded values, such as speed, torque and mechanical power consumption/generation. Other displayable values include slip, voltage, current consumption, power factor and efficiency. Finally, the toolbar allows for quick access to frequently used functions such as saving, settings, display characteristics and run/stop the machine.

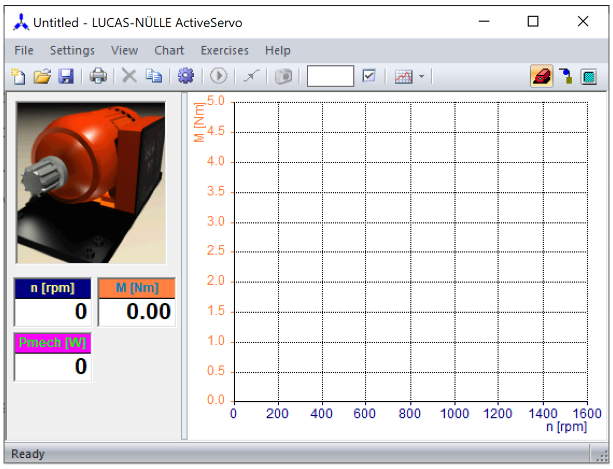
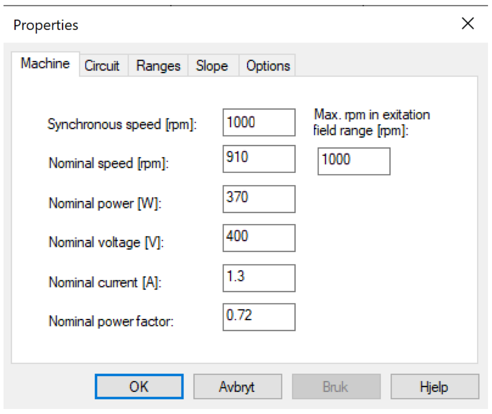


Figure 3.8: ActiveServo main window

The first step in initialising the program is to add the test machine's parameters into the properties window. The parameters needed are the various nominal speeds, power, voltage, current and power factor, as shown in Figure 3.9, as well as information about whether this is a multi-, single-phase or DC machine, its voltage- and current range and safety limits filed under the *Circuit*, *Ranges* and *Options* tabs. These parameters allow the program to tailor the brake force and calculate the appropriate responses based on the test motor's abilities. ActiveServo is explicitly created for the test stand, so the brake's machine info is already hardcoded into the program.



Synchronous speed [rpm]:	1000
Nominal speed [rpm]:	910
Nominal power [W]:	370
Nominal voltage [V]:	400
Nominal current [A]:	1.3
Nominal power factor:	0.72
Max. rpm in exitation field range [rpm]:	1000

Figure 3.9: ActiveServo properties window

One very convenient feature found in ActiveServo is the ability to record the machine characteristics of the test motor. During this test, the test motor is gradually braked in steps from its no-load speed down to a complete halt by the servo-brake, all while the torque and speed are recorded and plotted in the main window in Figure 3.8. The result is the machine characteristics speed torque plot, which provides crucial information about the machine's working range, up to the pull-out torque. In addition, the parameters braking process parameters can be specified to a various number of steps and starting points in the *slope* tab found in the properties window displayed in Figure 3.9, allowing for, to a certain degree, customised braking sequences. The sequences are initiated by the command *Switch drive on* found in *Settings* or pressing the *run*-button found in the toolbar and manually switching the drive to RUN using the button displayed as 11 in Figure 3.6.

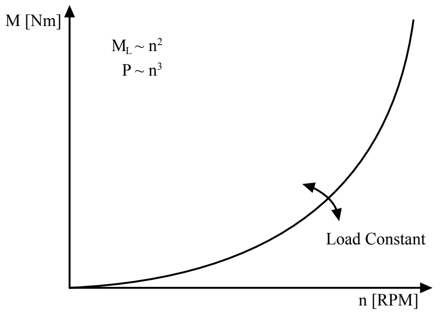
The machine parameter tab in ActiveServo also contains a speed-control mode, where the servo-motor is synchronised to the constant speed of the test motor. In this mode of operation, the test motor can be tested in all four quadrants (see Figure 3.7), which is particularly suitable for recording the characteristics of an asynchronous machine.

Using the machine characteristics plot, one can predict how the motor will handle the various load at various speeds. ActiveServo includes seven premade load simulation modes to demonstrate the effects of torque change, each affecting the braking torque and speed relation in different ways. The modes are found under settings when ActiveServo is in Load Emulation mode and their corresponding load constant is adjusted using the panel on the left hand.

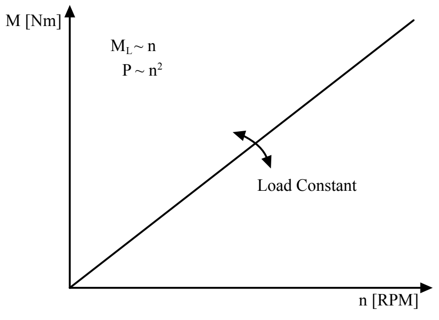
In the *Pump/Fan* mode, the brake resembles the torque response of a pump or a fan with a quadratic torque rise as the speed increases. The model assumes an ideal characteristic, ignores other factors such as friction and inertia, and lets one adjust a *Load Constant* parameter that resembles a unitless diameter of the pump or fan impeller—the higher the load constant, the larger the diameter and steeper torque curve. Equation (3.1) describes the torque curve seen in Figure 3.10a, where M is the torque, N the motor speed, $\dot{m}$ the flow and M_{max} the maximum possible torque, in this case, with a $0.37\,kW$ test motor, $10Nm$.

$$M = \left(\frac{\frac{N}{\dot{m} \cdot l_c}}{4000 \cdot 27} \right)^2 M_{max} \tag{3.1}$$

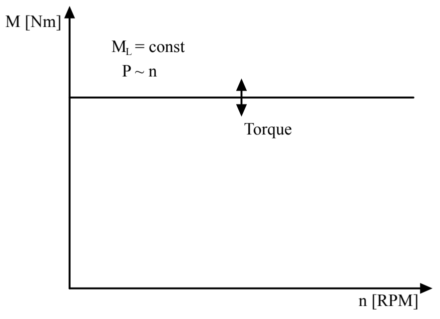
In the *Calander* mode, the brake simulates the torque response of a calander, a machine that presses cloth or paper through rollers. As Figure 3.10b shows, the brake torque increase is linear with the increase in speed. As in the *Pump/Fan* mode, the model ignores other variables such as friction and inertia, and the torque is calculated as in Equation (3.2).



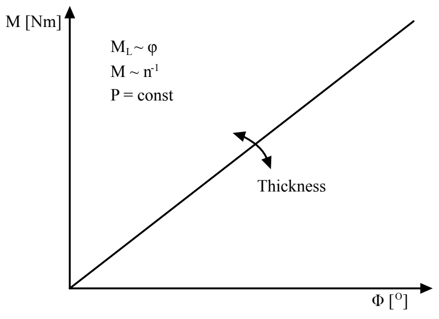
(a) Pump/Fan



(b) Calander



(c) Hoist Drive



(d) Winding Machine

Figure 3.10: ActiveServo Load modules speed/torque relations

$$M = \frac{\frac{n}{\dot{m} \cdot l_c}}{4000 \cdot 50} \cdot M_{max} \tag{3.2}$$

The *Hoist Drive* mode shown in Figure 3.10c is the simplest load model included and is used for loading the test motor with a constant torque over its whole speed range. In this mode, the torque is directly adjusted in the parameter dialogue box on the left side of the screen, and no load constant is used. This mode can operate in all four quadrants and can simulate a crane lifting or lowering a load.

The last model shown in Figure 3.10, the *Winding Machine* mode, emulates the torque of a winding machine where the torque increases linearly with the amount of wound turns in one direction. The increased torque comes from the simulated increased thickness of the wound material, which increases the offset of the pulling angle. If the machine rotates in the other direction, no counter-torque is applied, and the number of turns is reset to 0 if the numbers of turns in both directions are equal. This mode also corresponds to an ideal characteristic where friction and inertia are ignored. The adjustable parameter in the dialogue box is the thickness of the wound material, which translates to the Figure 3.10d line's steepness.

In addition to the models shown in Figure 3.10, ActiveServo has three additional modes, the *Compressor*, the *Inertia Wheel* and the *Time-dependent* load. The torque response corresponds to that of a reciprocating compressor (also known as piston compressor) with a pressure tank in the compressor mode, where each revolution increases the torque until the maximum torque is reached and the compressor discharges. During discharging, the torque rapidly collapses. The volume of the pressure tank is altered in the parameter dialogue, and a linear increase in torque can be seen due to the increasing pressure. If one attempts to rotate the motor in the reverse direction, it will be blocked by maximum torque, and great care must be taken to ensure the motor is wired and configured to run the correct way before powering it on. The *Inertia Wheel* mode, on the other hand, represents the response of an inertia wheel, or mass, with a lagging reaction. The test machine's response to starting and stopping various masses can be examined by adjusting the mass in the parameter dialogue box, ranging from 3 to 3000, where 1000 corresponds to $14000\,kg \cdot cm^2$. Friction can also be added from 1 to 100

Finally, the *Time-dependent* mode allows for a more customised load sequence and allows for rapid torque changes. The torque sequence can either be programmed using the built-in function that allows for dragging and adding multiple points to a premade, initially flat time-line, i.e. a constant torque, or by adding a series of comma-separated values using the *Import from clipboard* function. The selected values are then imported and plotted in equal x-intervals along the graph, and the program is started by pressing the *Run*-button.

Restrictions

As described in the sections above, ActiveServo enables quite a few albeit reasonably simple load-emulation models. However, it lacks the options of further model expansions and adaptations, appears somewhat restricted to its existing models and does not contain any control options for the test motor, as a separate frequency drive runs that (see Section 2.2). In other words, the communication solutions need to be revisited to include the test motor in the control circuit or expand the control interface to include external models made in other programming environments.

3.3 Communication Paths

The test stand and servo motor controller built by Lucas-Nülle described in Section 3.1 is based on a servo drive (see Section 2.3), model 9400 as seen in Table 3.1, made by the German company Lenze. The servo drive itself is a standalone component made for implementation in larger systems and not necessarily tailored for this specific test stand. The preconfigured communication consists of the USB port shown as number 12 in Figure 3.6, which allows for computer control using ActiveServo or manual control using the physical user interface. All the information received by the interface (either manually or through USB) is processed, translated and sent to the servo drive through a CAN-bus protocol, as sketched in Figure 3.11.

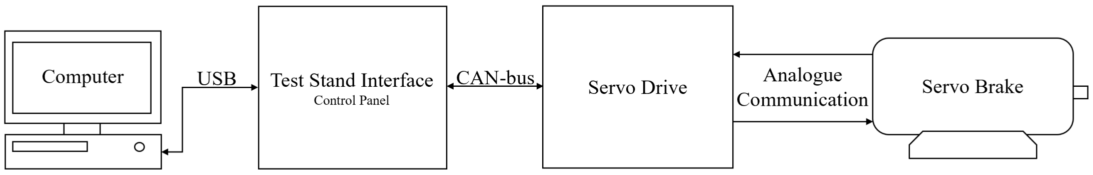


Figure 3.11: Communication from computer to drive

3.3.1 Serial Communication

In short, serial communication is a form of digital signal transmission used extensively within the computer- and electronics industry. Serial communication is often favoured over other communication solutions such as parallel interfacing or analogue controllers due to its simplicity and low hardware overhead [30]. Although there are many protocols within the serial communication category, their base working principle is the same: Sequences of digital signals sent one by one through a pair of wires. The sequence represents a message, where the structure is dependent on what protocol is used. Figure 3.12 shows an example of the character 'A' transmitted as the ASCII binary pattern 0100 0001 over a serial line using the popular RS232 protocol, where the line represents the state of the transmitted voltage level (low or high) [31].

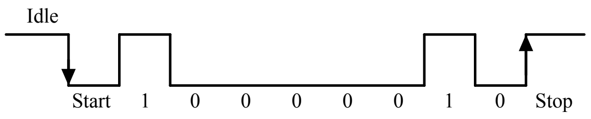


Figure 3.12: ASCII character 'A' using RS232 [31] (edited).

The main serial communication used on the test setup is as described in Figure 3.11 USB and CAN-bus, further elaborated in subsubsection 3.3.1 and subsubsection 3.3.1.

CAN-bus and CANopen

CAN-bus is a system used as a standard in almost all automotive industries, ranging from motorcycles to trucks to even ships and aeroplanes, for various reasons:

- The digital bus system allows for more straightforward wiring and cost-efficient installation than analogue signalling.

- The system is fully decentralised, enabling one point of entry for central diagnostics, data logging and configuration.
- CAN-bus system is *very* robust and efficient., as the signals are close to unaffected by electromagnetic interference and disturbances.
- All the frames are ID-prioritised, so the top priority frames are sent out on the bus without causing a disturbance, making CAN-bus ideal for safety-critical applications such as vehicle and machinery control.

The CAN-bus protocol provides the essential means of communication, but is a very simple messenger. Therefore, higher layer protocols have been created to implement more advanced functionality. As a result, several higher layer protocols are tailored to fit specific needs. For example, in the case of the servo drive used in this project, CANopen is used as the communication link between the control panel and the servo drive and the link between the drive's internal parts. [32]

CANopen is widely embedded in control applications such as industrial automation and is today extensively used in motor control. CANopen is a higher layer protocol based on CAN-bus, meaning it uses the same frame structure and 11-bit identifier. Figure 3.13 illustrates the structure of a standard CAN frame or message, where the numbers signify the bit size of each field.

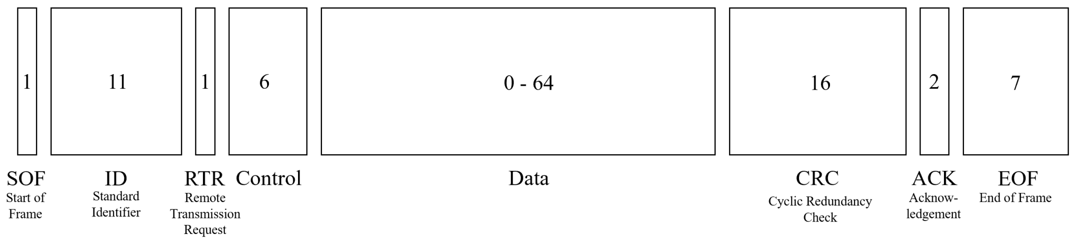


Figure 3.13: Standard CAN frame [33] (edited).

The frame starts with a *dominant 0* to signal a new message being sent. After comes the identifier, where lower values have a higher priority, followed by the RTR, which tells if there is a send message or a request for data from another node and the control field that contains the Identifier Extension Bit (IDE) and length of the data field that comes after. The data field includes the payload (data bytes) that can be extracted and decoded for information. Next, the CRC ensures data integrity after transmitting, and ACK acknowledges if the node has received the data correctly. In the end, the EOF signals the end of the frame. [33]

CANopen allows for a few more options, allowing for a few different communication models, such as controller/agent or client/server. In addition, a preset of encoded device states can be changed in agent devices by the controller, e.g. reset after a fault. Finally, object dictionaries, electronic datasheets, and device profiles enable CANopen devices to be integrated into more extensive, vendor-independent systems. The servo drive used in this project has a handy LED display seen in Figure 3.14 signalling the drive's state, which adds to a quicker troubleshoot if anything were not to work correctly.

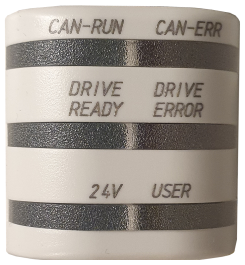


Figure 3.14: Lenze Highline 94000 Servo Drive Drive State indicator

The indicator can constantly display the drive's current state or errors through a series of blinking signals and colour codes thoroughly explained in the drive's manual[34].

Universal Serial Bus

The USB (Universal Serial Bus) is, without doubt, one of the most widespread and commonly known serial communication protocols there is today. For example, the servo motor control panel of the test stand uses USB to communicate with a computer and its program ActiveServo, further described in subsection 3.2.2. The protocol allows for fast data transmission over a two-wire (D+ and D-) cable controlled by clock pulses. The two remaining pins are a 5V pin used to power devices, and a ground pin. Figure 3.15 illustrates the pinout on an ordinary USB connector up to USB 2.0. In the newer USB 3.0 connectors, there is are 5 additional pins, in short allowing for a higher bandwidth and even faster transmissions.

The USB is, like CANbus, a serial protocol used where messages are transported one by one through the cable. A USB package can consist of up to six data packet fields, as shown in Figure 3.16. The package illustrated is a token package, where the sync field synchronizes the clocks from both the receiver and the transmitter, while the PID field provides information about what type of data is being sent. The four types of PID are tokens, data, handshake and special, which serve different purposes and determine the

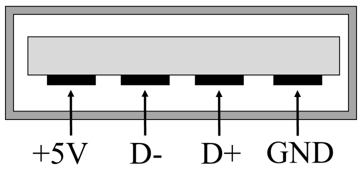


Figure 3.15: Standard USB pins up to USB 2.0 [35]

structure for the rest of the USB package. The address field includes the address of the device the package is sent to, while the endpoint specifies the endpoint number. Finally, the CRC, as in CANbus, checks if the data in the package is free of errors before the EOP field indicates the end of the packet. [36] [37]

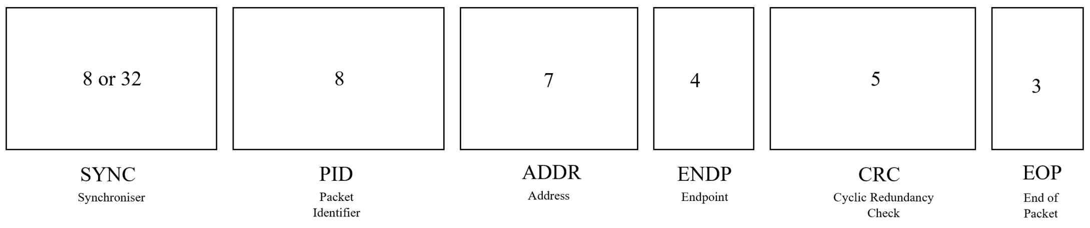


Figure 3.16: Standard USB token packet

In contrast to CANbus(unless using CANopen in controller/agent mode), USB is a host-centric bus, meaning the host initiates all transactions. The host is usually a computer, controlling various applications connected to it. In this project, the various applications are the servo drive control interface and a microcontroller, further elaborated in subsection 4.2.4. An essential factor to remember is that although one is used to multiple USB devices connected, the USB is still a form of serial communication. The sheer speed of the USB protocol allows for the illusion of multiple devices and packages being handled simultaneously, despite only one package is transferred at a time. [38]

3.3.2 Analogue Communication

In the early days, before the digital revolution and when processor speed and memory were too limited to support anything but discrete control functions, analogue signals were used for most control applications. The analogue control signals offer simple, universal units that are easy to implement in cross-brand systems. Every value is defined as simple physical SI-units such as voltage or current and requires no translators for byte-long packages. Due to their universal and straightforward nature, analogue control signals are still widely used in control applications today. In this project, both the VFD and Servo Drive allow for control by analogue signals, as seen in Figure 4.3a and Figure 3.5.

Standardised analogue control signals include 0-10V, +-10V and 4-20mA, where the control range or sensor range is scaled to match the control signal. First, the measured value (MV) is scaled to a value between 0-1, based on the measuring span and lower range value (LRV). Next, the scaled measured value (MV_{scaled}) is implemented into the equation for the control signal output CS_{out}, scaled similarly between its Lower Range Value CS_{LRV} and Upper Range Value CS_{URV}. The equation assumes linear changes. [39]

Table 3.3: Analogue control signals and common device applications

Signal	**Commonly used in**
4 - 20 mA	PLCs, Drives, Sensors.
0 - 20 mA	Sensors, Actuators.
0 - 5V	Microcontrollers and computers.
0 - 10V	PLC and Drives.
-10 - +10V	PLC and Drives.

Table 3.4: Analogue Digital Converters

Type	**Meaning**
ADC	Analogue to Digital Converter. Enables analogue reading by a digital controller.
DAC	Digital to Analogue Converter. Enables analogue writing by a digital controller.
ADC,DAC	Both of the above, often referred to as Analogue I/O-modules.

$$\begin{gathered} MV_{scaled} = \left(\frac{MV - LRV}{|URV - LRV|} \right) \\ CS = (|CS_{URV} - CS_{LRV}|) \cdot MV_{scaled} + CS_{LRV} \end{gathered} \tag{3.3}$$

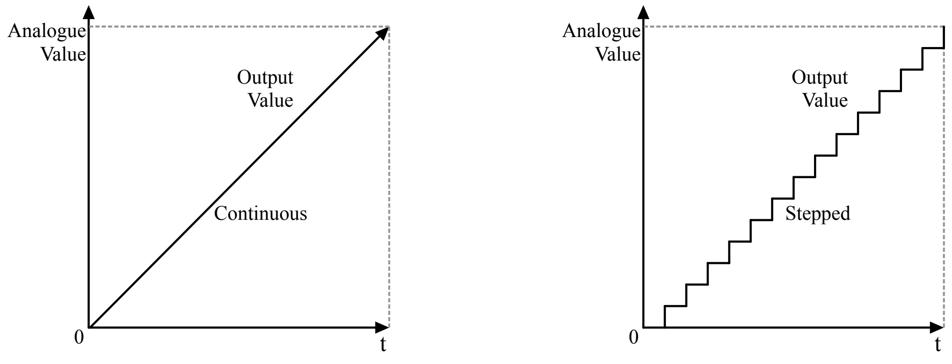
Computers, be it personal laptops, PLCs (Programmable Logic Controllers), or a microcontroller, only understand digital messages. Therefore, converters are needed to write or read any analogue control signals. Table 3.4 describes the three main categories of such converters.

ADCs essentially work by taking snapshots of the analogue value in one instant of time and converting it into a binary number representing the ongoing analogue level. While the analogue or "real" value has an infinite number of different levels, the digital versions have to work in steps, determined by their resolution. Figure 3.17a illustrates the difference between a linear analogue level in real life (3.17a and how the digital converter views it(3.17b. [39]

The converter's resolution determines the number of steps, as shown in Equation (3.4), where CS_{res} is the Control Signal resolution and b is the converter's number of bits.

$$CS_{res} = \frac{|URV - LRV|}{2^b - 1} \tag{3.4}$$

For example, an 8-bit 0-10V converter will have a resolution of $\frac{|10V - 0V|}{2^8 - 1} \approx 0.04V$.



(a) Continuous (Analogue) (b) Step-wise (Digital)

Figure 3.17: Increasing analogue value continuous and step-wise [39](Edited)

DACs are the opposite of the ADCs. They convert digital data into analogue values with output voltage or current proportional to the value of their digital input. The digital to analogue values are scaled similarly to the ADC described in Equation (3.3), where the digital range and the analogue values remain proportional, assuming a linear relationship. The output value increase will remain similar to the one illustrated in Figure 3.17b if the x-axis is the corresponding digital value and the y-axis represents the analogue output value. Hence, the higher resolution, the smaller steps and the more accurate control, as described in Equation (3.4). [40]

There are multiple methods to convert a digital signal to an analogue output. However, one of the most common methods found in low-cost applications is a Pulse-Width-Modulation (PWM) DAC [40]. PWM essentially works by switching a fixed DC voltage on and off in specific intervals, creating a high-frequency pulse. As seen in Equation (3.5), the relationship between the on-time (t_{on}) and off-time (t_{off}), named Duty Cycle (D), determines the output voltage, as described in Figure 3.18 and Equation (3.6), where V_{in} is the fixed input voltage and V_{out} is the mean output voltage. The PWM frequency in Equation (3.7) is calculated by dividing 1 by the total cycle time, t_{cycle}, measured in seconds. [41]

$$D = \frac{t_{on}}{t_{off}} \tag{3.5}$$

$$V_{out} = V_{in} \cdot D \tag{3.6}$$

$$f = \frac{1}{t_{cycle}} \tag{3.7}$$

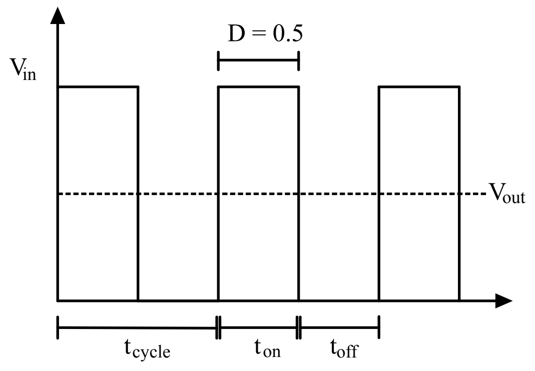


Figure 3.18: Pulse Width Modulation Voltages

Figure 3.18 shows an example of a PWM with a duty cycle = 0.5, which means the voltage is on half of the time, causing, as shown in Equation (3.6), resulting in a mean voltage and PWM output voltage of $0.5 \cdot V_{in}$.

An apparent disadvantage of PWM regulation is the chronic voltage variation, constantly switching between zero and maximum voltage. Firstly, this means the equipment connected to the DAC will be exposed to the maximum voltage level even at lower output voltages. Secondly, constant switching, especially at high frequencies, can cause electrical noise that affects the equipment if the cables are not shielded. Therefore, when using PWM as a voltage regulator, great care must be taken to ensure the equipment can handle the switching output, unless measures to reduce the switching voltage are taken. [42]

A simple method to reduce the sudden square voltage changes is to add an RC-filter, consisting of a resistor in series and a capacitor connected to ground, as shown in Figure 3.19.

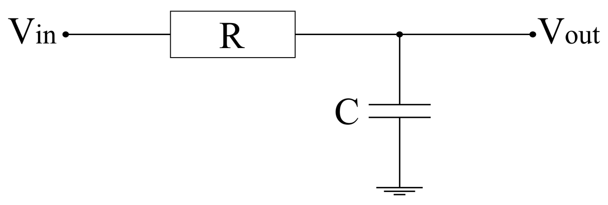


Figure 3.19: RC Low Pass Filter

The filter smoothes the voltage ripples, creating a less noisy, more stable voltage and works according to the principles shown in Equation (3.8). The output voltage V_{out} depends on the relationship between the resistor R and the reactance X_C of the capacitor C. [42]

$$V_{out} = V_{in} \cdot \left(\frac{X_c}{\sqrt{R^2 + X_C^2}} \right) \tag{3.8}$$

The reactance is calculated using Equation (3.9), where f is frequency and C the capacitance.

$$X_C = \frac{1}{\sqrt{2\pi f C}} \tag{3.9}$$

Providing approppriate resistive and capacitive values to an RC low pass filter will at medium frequencies result in an output similar to the one shown in Figure 3.20, where the capacitor charges during the t_{on} and discharges during t_{off}, reducing the voltage difference between the two states. [43]

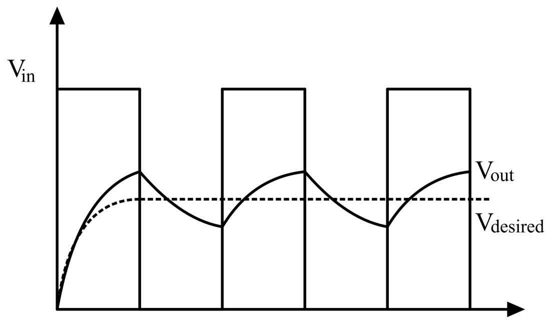


Figure 3.20: RC Low Pass Filter output with PWM input

4 Implementation

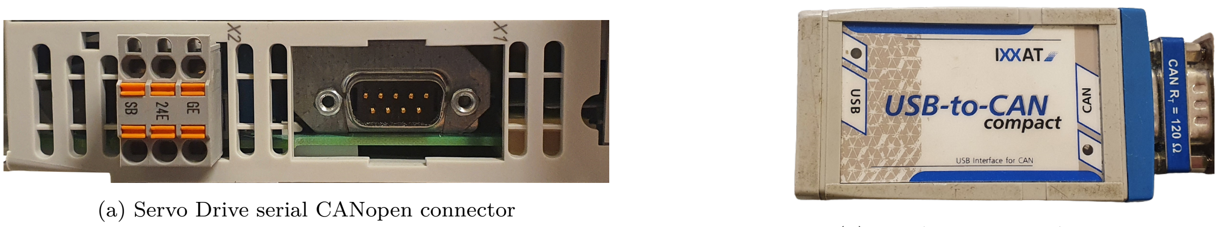
The servo drive test stand allows, as elaborated in subsection 3.2.2, for either manual control by a physical user panel or through its corresponding program ActiveServo. These allow essential control of speed and torque and can, in elementary manners, simulate a few applications such as a pump or a hoist drive. However, other customised control options are not possible using the current interfaces. Therefore, the communication paths and control interfaces must be revisited to expand the stand's application areas.

4.1 Servo Drive serial communication

As the test stand is to be used with computer run simulation tools, the intuitive solution is to utilise the existing USB connection found in Figure 3.6, used as the communication path to ActiveServo. However, to prevent any unintentional use, the USB driver for the control interface seems to be restricted only to cooperate with the ActiveServo-program. As a result, other programs cannot read and decode the serial packages (described in subsubsection 3.3.1) sent from the control interface, leaving the option of adapting other simulation tools to control the interface through the existing USB port out.

The servo drive's communication with the control interface happens through the CANopen protocol, connected by an easily accessible serial port marked *X1* found on the drive's side panel, as shown in Figure 4.1a. This port allows for direct communication with the servo drive without having to go through the additional external control interface.

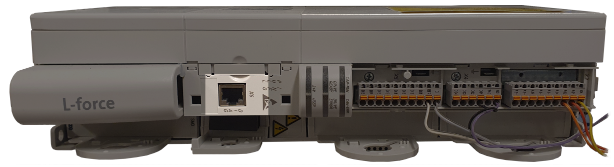
To access the CANopen drive interface, any standard CANconnector would, in theory, work as CANopen is a higher layer protocol of CAN. The Ixxat USB to CAN adapter



(a) Servo Drive serial CANopen connector

(b) IXXAT USB to CAN adapter

Figure 4.1: Lenze 9400 servo drive CANbus connection


(a) Diagnostics port location

(b) Diagnostics USB adapter

Figure 4.2: Lenze 9400 servo drive diagnostics port

shown in Figure 4.1b was used to connect the drive to a computer. A compact, portable adapter acts as a translator between the USB and CAN protocol described in subsection 3.3.1. There is a 120 Ω terminal resistor between the CANhigh and CANlow pins on the adapter's right-hand side to mimic the bus terminator found on each end of a physical CAN-network. [44]

The adapter is connected between the X1 port and a USB port on the computer, and run through a configuring program named Lenze Engineer (further described in subsection 4.3.4), a program made for configuring this series of Lenze servo drives. Unfortunately, this program and drive are restricted exclusively to accept on-brand adapters, resulting in the Ixxat adapter only being able to read incomplete messages, rendering the data unreadable. Lenze does provide an on-brand CANbus to USB adapter [45], but due to economic reasons, the purchase of such an adapter was not prioritised in this project. The remaining option for serial communication between the drive and a computer is the diagnostics port marked as $X6$, located at the front of the servo drive and highlighted in Figure 4.2a.

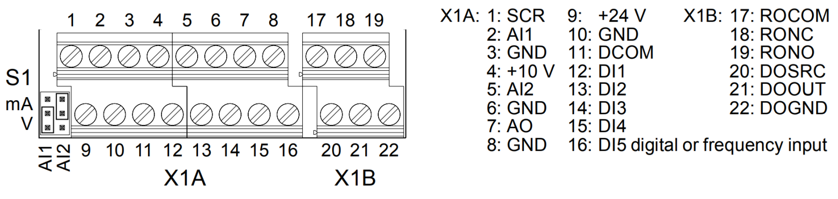
The diagnostics port is not made for permanent control signals but for simple configurations and drive maintenance, such as troubleshooting faults and firmware updates. To access the diagnostics port, one needs the diagnostics adapter pictured in Figure 4.2b, running an rj50 cable to the drive and a USB plug to the computer. When connected, the program Lenze Engineer (subsection 4.3.4) is used for configuring the drive. However, the diagnostics adapter does not allow for extensive real-time drive control, leaving the only viable option for the final communication module, the analogue I/Os.

4.2 I/O Modules

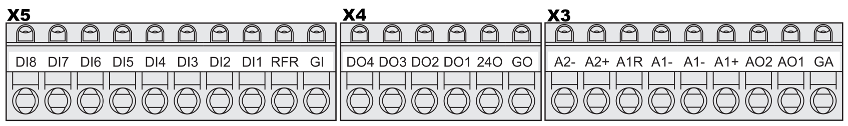
The servo drive and the VFD both include analogue I/O modules, which, as described in subsection 3.3.2, rely purely on analogue voltage- and current levels and are not vendorspecific. The I/O module's simple nature makes them ideal for combining multiple cross-brand devices into one system. This section describes the I/O modules found on the VFD and the Servo Drive, as well as the three modules for USB (computer) to analogue I/O considered for this project.

4.2.1 Existing setup

As mentioned in subsection 4.2.1, the VFD and the servo drive come with preinstalled I/O-modules, allowing for communication through analogue signals. The I/Os are, to a certain extent, programmable as to which values they control. The VFD has multiple pre-programmed presets with various IO configurations, the configuration named *Torque Control* allows for both speed and torque adjustments, with the functions of each I/O terminal described in Table 4.1. The functionality of the servo drive's I/O terminals found in Figure 4.3b are not tied to fixed functions like the VFD's terminals in Figure 4.3a but can be configured to various settings using the manufacturer's software Lenze Engineer, further described in subsection 4.3.4 [46].



(a) ABB Variable Frequency Drive [47]


X5
X4
X3
DI8 DI7 DI6 DI5 DI4 DI3 DI2 DI1 RFR GI
DO4 DO3 DO2 DO1 24O GO
A2- A2+ A1R A1- A1- A1+ AO2 AO1 GA

(b) Lenze Servo Drive [48]

Figure 4.3: I/O terminals on test setup

Table 4.1 describes the analogue ranges for the various terminals. Conversion from reference function value to analogue voltage setpoint and vice versa is shown in Equation (3.3).

Reading and writing to the setup's I/O terminals requires an interface able to translate serial communication USB to corresponding analogue voltages. The three attempted versions of DAQs are described in subsection 4.2.2 toi subsection 4.2.4.

Table 4.1: I/O terminals

	Terminal	**Range**	**Alt. range**	**Function**
ABB ACS350 VFD				
	AI 1	0-10 V	±10 V	Speed SetPoint
	AI 2	0-10 V	0-20 mA	Torque SetPoint
	AO	4-20 mA	-	Speed
	DI 1	24 V	-	Direction
	DI 2	24 V	-	Speed / Torque mode
	DI 3	24 V	-	Preset const. speed 1
	DI 4	24 V	-	Ramp Pair Selection
	RO	24 V	-	Fault
	DO	24 V	-	Fault
Lenze 9400 Servo Drive				
	AI 1	±10 V	±20 mA	Programmable
	AI 2	±10 V	-	Programmable
	AO 1	±10 V	-	Programmable
	AO 2	±10 V	-	Programmable
	RFR	24 V	-	Controller Enable
	DI 1...8	24 V	-	Programmable
	DO 1...4	24 V	-	Programmable
	24O	24 V	-	Voltage supply

4.2.2 National Instruments DAQ

The American company National Instruments has a product line specifically for data acquisition (DAQ) systems, i.e. measuring physical phenomena, converting them to computer-friendly values and utilising them through programmable software. The DAQ devices include the NI-DAXmx driver software and are fully integrable with LabVIEW environments and languages such as Python, ANSI C, and Visual C#. Figure 4.4 shows the NI CompactDAQ 9174 chassis with the NI9201 and NI9263, analogue input and output modules made for -10 to +10V.

The NI DAQ instruments all come with NI MAX software, made for easy implementation and straightforward testing of the equipment. The module is connected to the computer through a USB and runs through the NI MAX. The analogue outputs can, through this program, write value out for each channel with impressive accuracy. The program can also read analogue inputs and show the results in a live plot. If the chassis contained digital out- and input modules, the same actions could be performed with them. However, to implement the hardware in a more automated system, one would have to add it to another program, for example, a Python script. To implement any NI DAQ hardware

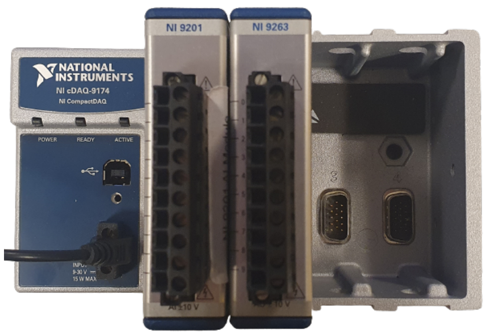


Figure 4.4: National Instruments compact Data Acquisition

into a Python script, functions from `nidaqmx` library can be used to access the terminals directly. Below is an example of using the nidaqmx Task to read values using the `nidaqmx` library:

```
import nidaqmx
with nidaqmx.Task() as task_read:
    task_read.ai_channels.add_ai_voltage_chan(terminal)
    value = task_read.read(number_of_samples_per_channel=1)
    return value
```

Figure 4.5: NI DAQ serial communication Python task

These premade functionalities within the library create a very straightforward access to the I/O terminals, setting up for a fast and easy I/O control. The downside of these DAQ systems from National Instrument is their cost. None of their simpler models has the required I/O terminal configuration to satisfy the needs shown in Table 4.1. The only viable option is the compactDAQ shown in Figure 4.4, which unfortunately is not within the university's budget to remain at this test stand.

4.2.3 PicCircuit microcontroller

A much more budget-friendly solution to an I/O-module is a microcontroller. Used for a previous project, the university had access to a PicCircuit iCP12 USB stick, shown in Figure 4.6. This is a small, very simple USB IO Board in microcontroller format, capable of reading and writing analogue and digital values up to 5V. The board comes with corresponding software similar to NImax, albeit more straightforwardly without too many settings available. [49]

Unfortunately, this board only contains two analogue outputs, as opposed to the four needed (see Table 4.1) to run the test stand. In addition, it has no form of premade libraries available for further expansion, e.g. Python scripts or any other language, making

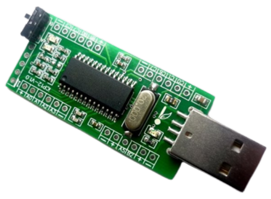


Figure 4.6: PicCircuit iCP12 USB stick [49]

it challenging to implement in a larger system. However, a microcontroller that *does* contain enough analogue outputs and can communicate with Python scripts is the Arduino Uno described in subsection 4.2.4. [50]

4.2.4 Arduino

Arduino is well-known open.source electronics platform consisting of both hardware and software. Arduino boards are relatively inexpensive microcontroller-based boards that are able to read from inputs and write to outputs in a similar fashion to the iCP12 stick described in subsection 4.2.3. However, the Arduino boards has the significant advantage of being fully programmable down to every pin and every bit transmitted back and forth to them. In addition, they are cross-platform compatible, with both the software and hardware being open-source and extensible. The programming of an Arduino board is done through the Arduino IDE, using a programming language much similar to C++, named simply the *Arduino Programming Language* and further described in subsection 4.3.2. [51]

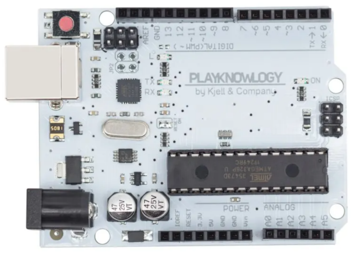


Figure 4.7: Off-brand Arduino Uno board [52]

The chosen board, shown in is a fully compatible knock-off version of the most popular Arduino board, the Arduino Uno [52]. For simplicity, the board will from here on be referred to as *the Arduino.* Arduino Uno is considered the standard Arduino board and contains 14 digital I/O pins, out of which 6 provide PWM output and six analogue input pins. All the digital pins can be configured as either output or input pins. The board operates at 5V, with the option of powering through an external power supply or using the power provided by USB, described how in subsubsection 3.3.1. If using only the USB as power, one must ensure that the board's power consumption does not exceed 500 mA, as that will break the internal fuse added to protect the computer's USB port. [50]

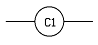
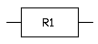
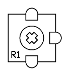
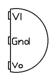
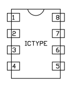
The Arduino will not be used to store any data but will act as an I/O-module, translating and transporting values back and forth between the test stand setup and the computer. However, the Arduino's output and input voltage range are $0-5V$, unlike the $4-20mA$, $0-10V$, $10V$ and $24V$ used by the servo drive and VFD (see Table 4.1). Therefore, a few converters are necessary to add between the test stand and the Arduino to combat the voltage difference. subsection 4.2.5 describes these custom-designed converters.

4.2.5 Analogue signal converters

The Arduino board is based on the typical $5V$ USB power and will need external amplifiers, reducers, and relays to correctly communicate with the test stand. Table 4.1 describes the voltage or current needed for each terminal, and this section goes through the converters made to fit the Arduino to meet those needs.

All the converter circuits are first designed as standard schematics and tested using the online electronics simulation tool CircuitLab [53]. After confirming the circuit works, the stripboard layout is designed manually and illustrated using the CAD (Computer-Aided Design) software AutoCad [54], where the circuit design and the components' physical size determine their spacing and board placement. Table 4.2 contains the description of all components' symbols.

Table 4.2: Stripboard circuits symbol description

Symbol	**Description**
🟩	Control signal wire
🟥	Power supply wire
—	Ground wire
⊗	Wire break
	Capacitor
	Resistor
	Potentiometer
	Voltage Regulator
	IC (size varies)

A generic 17W AC-DC converter [55] producing a DC output voltage of +- 15W powers the circuits' ICs and various reference voltages. In addition, all stripboard layouts for circuits containing op-amps are based around the IC op-amp LN324-N containing four independent channels with a pinout configuration as displayed in Figure 4.8.

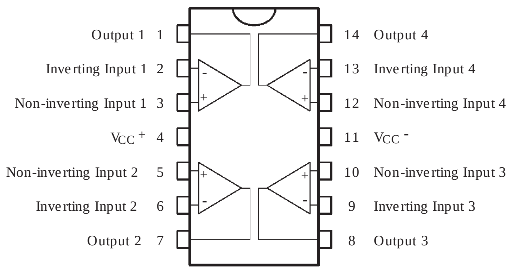


Figure 4.8: Operational Amplifier IC Lm324-N pinout diagram. [56]

Ouput voltage converters

As the VFD and the servo drive use $0-10V$ and $\pm 10V$ as analogue input values, the Arduino's $0-5V$ output voltage needs to be amplified before being sent to the devices. This section describes the process from the Arduino PWM output pins to the VFD and servo drive's analogue input terminals.

First, the PWM output voltage should go through a low pass filter similar to the one described in Section 2.2 to reduce the voltage ripples. The low pass filter is a low-cost and straightforward way to create a more stable voltage, allowing for more accurate control. It cannot produce a perfectly flat voltage output but is sufficient for this control circuit. The filter consists of a $47k\Omega$ resistor in series and a $1\mu F$ capacitor connected in parallel to ground. These values are chosen based on Equation (3.8), creating a low ripple, albeit reasonably slow response time due to its high resistance. The slow response should however not cause any significant disadvantage compared to the motors' inertia.

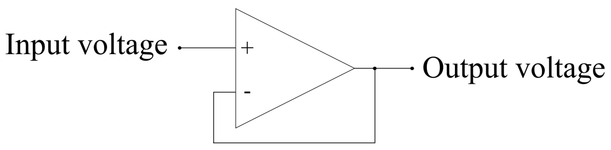


Figure 4.9: Voltage Follower

After the low pass filter, the filtered signal is sent through an operational amplifier(op-amp) configured as a voltage follower. The voltage follower is added because the low pass filter used is passive, causing any component added to the circuit to affect the filter's characteristics without a buffer. For example, the configuration shown in Figure 4.9 offers an op-amp where the output is connected to the input, which forces the op-amp to adjust its output voltage to equal the input voltage. Hence, the output voltage ”follows” the input voltage and avoids any retroactive influence on the filter connected to its input. [57]

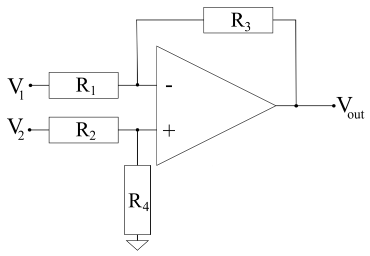


Figure 4.10: Op-Amp Differential Amplifier

From the voltage follower, the signal is sent through another op-amp, configured as a differential amplifier. In short, the op-amp multiplies the difference with a factor determined by the resistors. If $R1 = R2$ and $R3 = R4$, the output voltage in Figure 4.10 can be calculated as shown in Equation (4.1). [58]

$$V_{\text{out}} = \frac{R_3}{R_1} \cdot (V_2 - V_1) \tag{4.1}$$

Voltage doublers. The $0-5V$ to $0-10V$ converters allow the Arduino’s $0-5V$ outputs to communicate with the VFD’s $0-10V$ analogue inputs. The circuit illustrated in Figure 4.11 is created by combining the low pass filter, voltage follower, and differential amplifier. Here, the resistors R1 and R2 from Equation (4.1) consist of a connected resistor and potentiometer (R3+R6 and R1+R7, respectively). The potentiometers allow resistance adjustments for calibration even after the circuit is soldered, and their ideal value is calculated as the mid position.

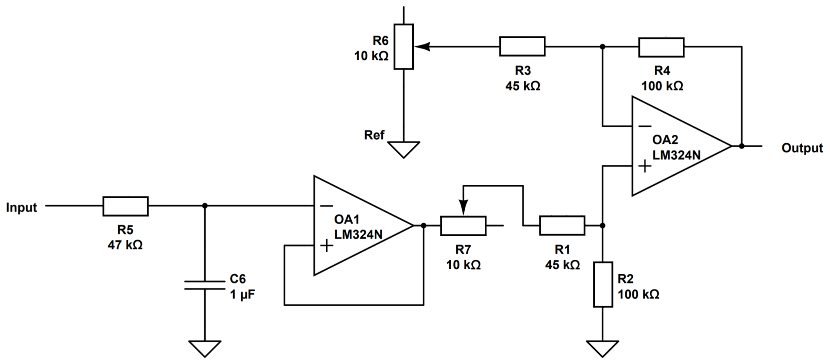


Figure 4.11: 0-5 v to 0-10 V converter circuit

This differential amplifier uses ground as its reference voltage and a resistance ratio that based on Equation (4.1) results in the voltage amplification:

$$V_{\text{out}} = \frac{R_4}{R_3 + \frac{R_6}{2}} \cdot (V_{\text{input}} - 0) = \frac{100k\Omega}{45k\Omega + \frac{10k\Omega}{2}} \cdot V_{\text{input}} = 2 \cdot V_{\text{input}}$$

Thus resulting in a linear amplification circuit with a low pass filtered input, where $V_{\text{ouput}} = 2V_{\text{input}}$, i.e. a voltage doubler. The filtered voltage doubler circuit is placed between the PWM output pins 4 and 6 and the VFDs analogue input terminals AI1 and AI2. The VFD will receive double the PWM voltage these Arduino pins outputs.

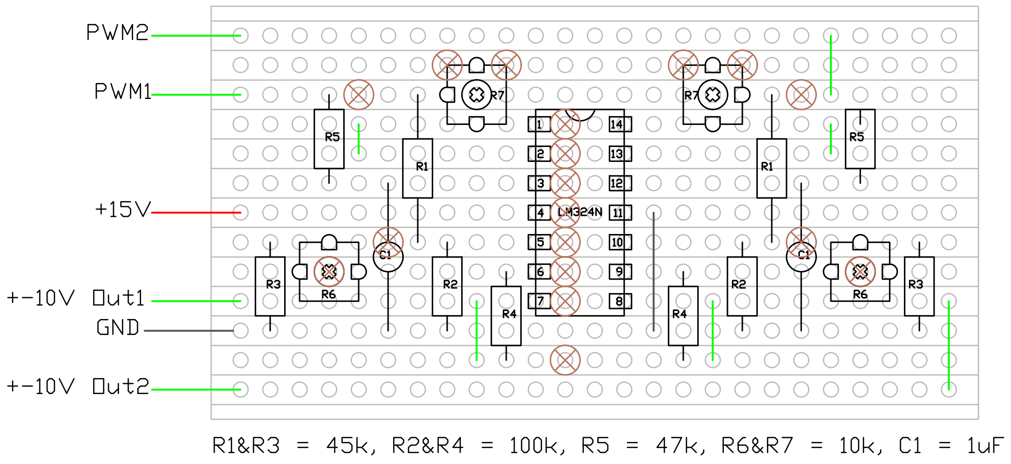


Figure 4.12: 0-5 v to 0-10 V converter stripboard layout

As the setup needs two voltage doublers, and each voltage doubler circuit needs two op-amps, one four-channel lm324-N IC should ideally cover the needs for both voltage doubler circuits. The components are all pin-mounted and 1/4w rated, and the 15V power supply is connected directly to the IC's positive power input. Figure 4.12 pictures the stripboard layout, with one amplifier mirrored on each side of the IC. All wires connecting to the power supply, the microcontroller and the VFD are placed on the side to improve wire management.

Voltage quadrupler. The $0-5V$ to $\pm 10V$ converters allow the Arduino's $0-5V$ outputs to communicate with the servo drive's $\pm 10V$ analogue inputs. This converter is based on the same principles as the voltage doubler, the most noticeable difference being its other negative voltage range. The circuit shown in Figure 4.13 contains the same low pass filter and differential amplifier as the voltage doubler, albeit with different resistor values and the reference voltage. For its reference voltage, the circuit uses a potentiometer as a voltage divider powered with 5V from a voltage regulator to create an input of $2.5V$, which is further sent through a voltage follower. With a reference voltage of $2.5V$ and an input voltage ranging from $0-5V$, the differential amplifier senses a difference between $-2.5V$ and $2.5V$.

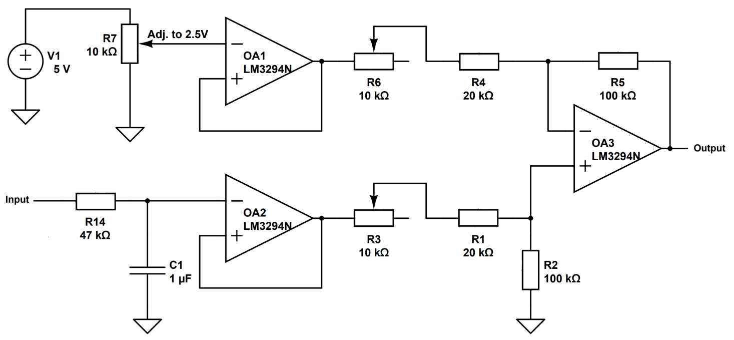
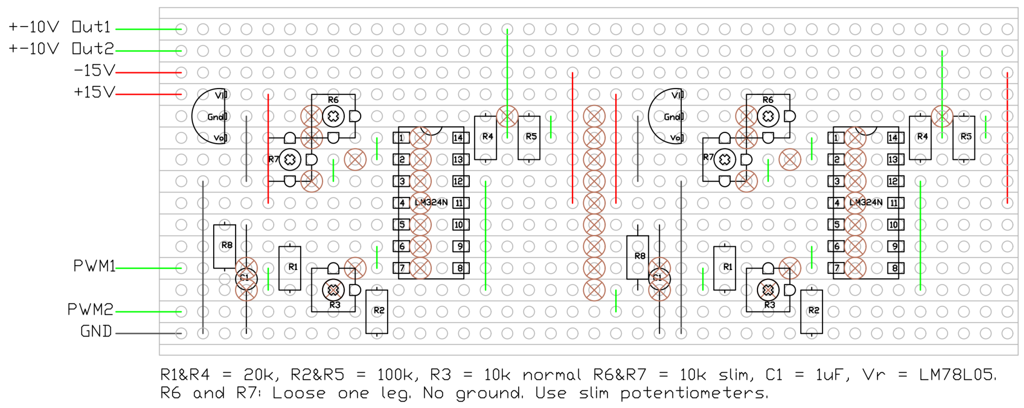


Figure 4.13: $0-5V$ to $10V$ converter circuit

By choosing resistive values to create an amplification of $4 \cdot V_{\text{input}}$, the circuit's output voltage will reach a range from $-10V$ to $+10V$. The circuits amplification is calculated using Equation (4.1) accordingly:

$$V_{\text{out}} = \frac{R_5}{R_4 + \frac{R_6}{2}} \cdot (V_{\text{input}} - V_{\text{ref}}) = \frac{100k\Omega}{20k\Omega + \frac{10k\Omega}{2}} \cdot (V_{\text{input}} - 2.5V) = 4 \cdot (V_{\text{input}} - 2.5V)$$

The stripboard layout displayed in Figure 4.14 shoes two voltage quadruplers based on their IC op-amp lm324-N as shown in Figure 4.8, built using the same techniques as the voltage doubler board in Figure 4.12. The most noticeable features separating them are the need for an additional IC and the two voltage regulators. The voltage regulators reduce the 15V input voltage down to $5V$ before using a potentiometer as a voltage divider to create the $2.5V$ reference voltage.



+-10V Out1
+-10V Out2
-15V
+15V
PWM1
PWM2
GND
LM324N
R1&R4 = 20k, R2&R5 = 100k, R3 = 10k normal R6&R7 = 10k slim, C1 = 1uF, Vr = LM78L05.
R6 and R7: Loose one leg. No ground. Use slim potentiometers.

Figure 4.14: 0-5 v to ±10 V converter stripboard layout

Voltage reducer The $\pm 10V$ to $0-5V$ converters allow feedback from the servo drive's analogue outputs to be read by the Arduino's analogue input pins. The converter bases its voltage transformation on the same principles as the amplifiers but with the resistor values reversed to reduce the voltage instead of amplifying it.

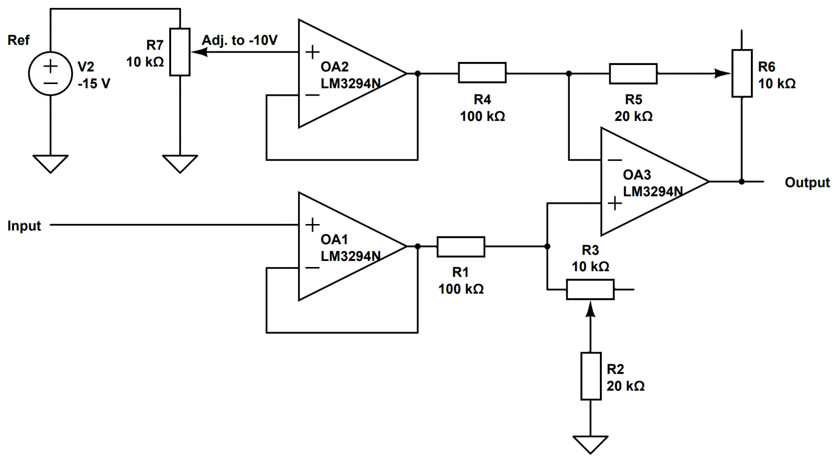


Figure 4.15: ±10 V to 0-5 v converter circuit

In this case $-10V$, the reference voltage is achieved by a potentiometer voltage divider receiving the $-15V$ from the generic power supply. In addition, the input low pass filter used in the two previously described amplifiers is omitted, as the servo drive outputs levelled DC voltage. The circuit's voltage reduction can be calculated using the same Equation (4.1) as the amplifiers:

$$V_{\text{out}} = \frac{R_5 + \frac{R_6}{2}}{R_4} \cdot (V_{\text{input}} - V_{\text{ref}}) = \frac{20k\Omega + \frac{10k\Omega}{2}}{100k\Omega} \cdot (V_{\text{input}} - (-10V)) = \frac{1}{4} \cdot (V_{\text{input}} + 10V)$$

which results in the voltage read by the Arduino essentially being a quarter of the output voltage if the reference voltage -10 is perceived as 0.

As illustrated in Figure 4.16, the voltage reducers' stripboard layout resembles the voltage quadrupler (Figure 4.14) in many ways. The stripboard contains two circuits, each based around three out of the four op-amps in each IC. The layout neither contains any low pass filter nor voltage regulator in contrast to the quadrupler, and their reference voltage is adjusted using R7 to -10V.

Current to voltage converter

The 4-20mA to 1-5V voltage converter allows feedback from the VFD's analogue output to be read by the Arduino's analogue input pins. This converter is the simplest of the

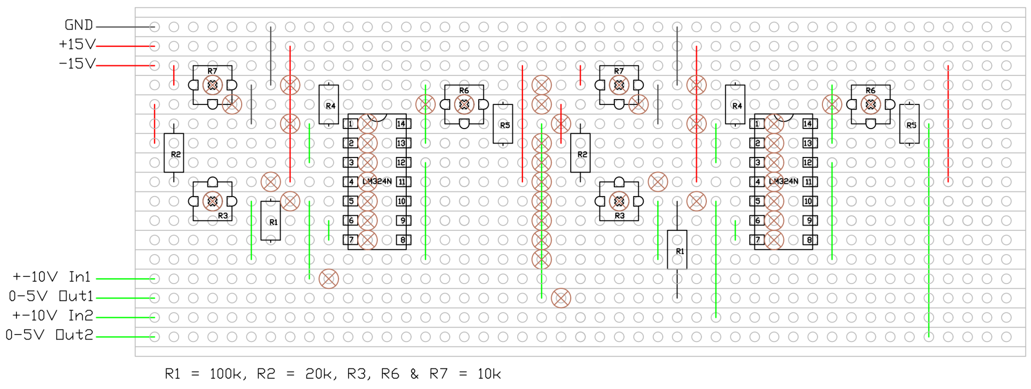


Figure 4.16: ±10 V to 0-5 v converter stripboard layout

ones created for this setup and consists in all its simplicity of four resistors, one of them a variable resistor, as illustrated in Figure 4.17.

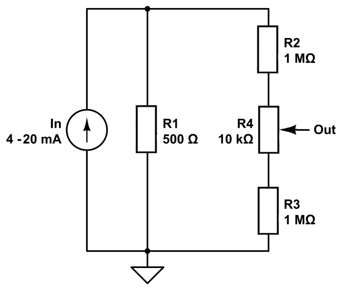


Figure 4.17: 4-20 mA to 0-5 v converter circuit

The converter consists of one 500ohm resistor parallel with two $1\,M\Omega$ and a $10\,k\Omega$ potentiometer. The high resistance in parallel with the 500ohm resistor causes the total resistance to be close to $500\,\Omega$:

$$R_{\text{tot}} = \frac{R_1 \cdot R_2}{R_1 + R_2} = \frac{500\,\Omega \cdot (1\,M\Omega + 1\,M\Omega + 1\,k\Omega)}{500\,\Omega + (1\,M\Omega + 1\,M\Omega + 1\,k\Omega)} = 499.875\,\Omega$$

Using ohm's law results in a voltage of $9.998\,V$ measured over the parallel resistors. Thius, the potentiometers' middle position results in an output close to 5V when the maximum current of 20mA is supplied: [59]

$$V_{\text{out}} = \frac{1\,M\Omega + (0.5 \cdot 1\,k\Omega)}{1\,M\Omega + 1\,M\Omega + 1\,k\Omega} \cdot 9.9975\,V = 4.999\,V$$

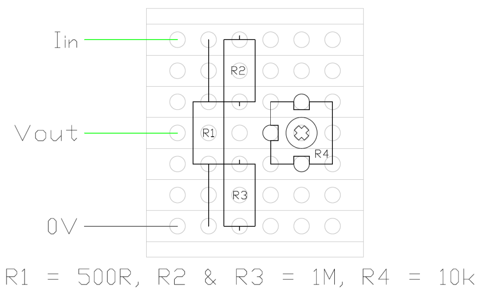


Figure 4.18: $4-20mA$ to $0-5V$ converter stripboard layout

By adjusting the potentiometer, it is then possible to convert the $20mA$ to 5V. Using the same procedure for the lowest 4mA VFD output gives a voltage of approximately $1V$, which results in a voltage input range of $1-5V$. This passive method consists of few elements and is simple to implement, albeit not the most accurate. It is, however, considered satisfactory for the current application.

The converter is built on a small stripboard as shown Figure 4.18.

Digital terminals

The digital terminals of the ABB VFD and the Lenze servo drive use 24V as input voltage. Conveniently, both devices contain a $24V$ output terminal, where combining this with relays allows for control by the Arduino. The relays used are Fujitsu Takamisawa *A5W-K* miniature relays with nominal voltage of 5V and nominal current draw of $28mA$, ideal for the Arduino outputs. As seen in Figure 4.19, the relays are to be controlled by digital Arduino output pins, V_{contr} with GND_{contr} as ground, while the $24V$ is supplied by the $24V$ outputs on the VFD and servo drive. The relays are double, but only one side is used to feed the digital terminals.

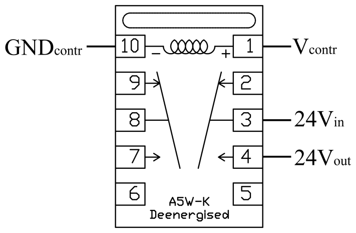


Figure 4.19: A5W-K relay [60](edited)

As seen in Figure 4.20, the relays are wired onto stripboards to save space and keep the wiring organised. In addition, the four relays are divided into two groups, keeping the $24V$ VFD and servo drive voltage sources separated.

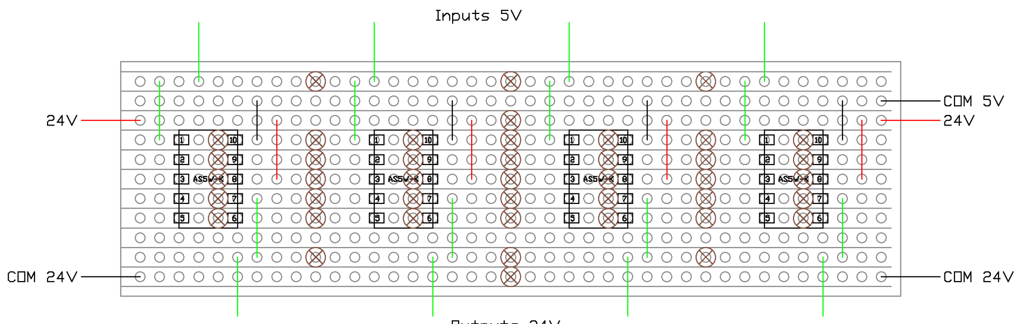


Figure 4.20: Relay stripboard layout

4.3 Interface programming

The programming employed on this interface is twofold and consists of an Arduino microcontroller sketch and a Python based control script. The Arduino sketch can be viewed as the created I/O module's firmware and is not to be changed once completed and uploaded, as it merely acts as an intermediary device and a translator between the Python script and the analogue values. On the other hand, the Python script contains all the logic and commands used to control the devices. Figure 4.21 illustrates a simplified version of the control signal path and its communication media.

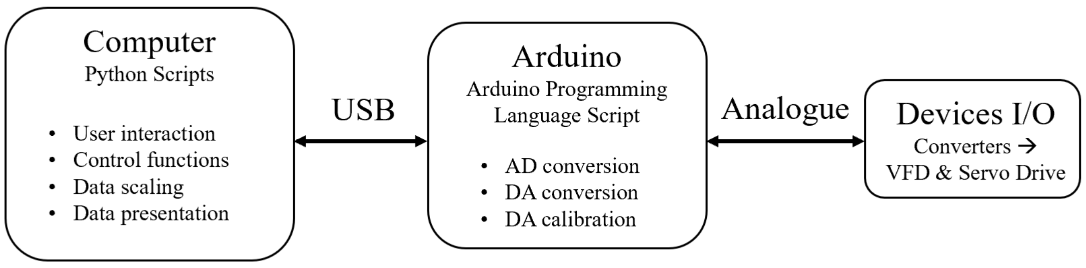


Figure 4.21: Communication between hardware and software

Before going into details, the key to understanding the process shown in Figure 4.21 is to know how the USB transmits data between the Arduino and the Python script. As described in subsubsection 3.3.1, the USB is a serial communication protocol, meaning it can only send one message at a time, albeit with a speed that enables the illusion of it transmitting multiple messages simultaneously. The same goes for the USB communication between the computer and the Arduino. Therefore, subsection 4.3.1 describes the data transmission structure between the Python program and the Arduino, while subsection 4.3.2 and 4.3.3 detail how the two scripts build, receive, and treat the messages.

4.3.1 USB transmission structure

All communication between the Python script and the Arduino happens through the USB, using functions created for serial communication. However, one weakness in Arduino's serial read functions is the excessive use of timeout-based solutions to determine the end of a serial message consisting of multiple elements, for example, a string. This weakness can lead to issues when attempting to read, e.g. something as simple as a number with multiple digits, as Arduino perceives it as not one multi-digit number but multiple one-digit numbers. When reading from the serial buffer into a String, the Arduino waits until the read function times out to determine if the string has ended. The timeout is often set to 1000ms, which is too slow for most control-related purposes and leaves the Arduino occupied even after receiving the entire message. One can reduce the timeout duration to minimise the dead time, but that leaves the risk of ending the read function too early and hence end up with only partial messages received. Therefore, to avoid the timeout issue, the serial communication from the Python script to the Arduino in this setup is configured to send and accept only one character at a time, whereas Arduino manually stores them in an array until a terminating character is received.

Most messages sent from the Python script to Arduino are based around three individual characters sent one by one, separated by the character '`x`' and ended with a termination character, chosen to be '`~`'. A typical example of a message structure from Python to Arduino is therefore:

```
axbxc~
```

Figure 4.22: Python to Arduino message example

In these messages, the first character, in Figure 4.22 '`a`', tells Arduino the primary intent behind the command, whether to write values to the output pins, calibrate its maximum values or any of the other functionalities described in Table 4.3 and subsection 4.3.3.

Table 4.3: Python to Arduino 1st character commands

1st Character	Command
e	Read all analogue inputs
o	Write to analogue output
p	Write to digital ouptut
q	Calibrate analogue output values
h	Switch built-in LED on
i	Switch built-in LED off

The second character, '`b`' in Figure 4.22, decides which channel number the message applies. For example, '`ax2`' as the initial characters tells the Arduino to write *something* to analogue channel number two. That *something* is decided by the final character 'c',and requires a more thorough explanation to comprehend fully.

As the Arduino cannot read strings without a function including timeout, as stated previously, multi-digit numbers can lead to quite a headache when wanting to write to analogue outputs. As a solution, numbers can be converted into single characters using their corresponding ASCII characters and sent as a single byte. ASCII code is the numerical representation is a character and was originally created to represent characters as numbers due to computers only understanding numbers [61]. However, as the ASCII table only contains a limited number of characters, the numerical range will be limited. All numbers to be sent as an ASCII character are therefore scaled to integer percentage values as a workaround, meaning all values sent from Python to Arduino is within the range $0-100$. When Arduino receives the percentage value encoded as an ASCII character, it decodes it back to its integer value and scales it to fit the analogue output values, which in the 8bit PWM outputs are in the range between 0 and 255. The scaling is done using the process described in subsection 3.3.2. Figure 4.23 shows an example of the conversion process where the setpoint of a nominal $910\,rpm$ motor is set to $405\,rpm$.

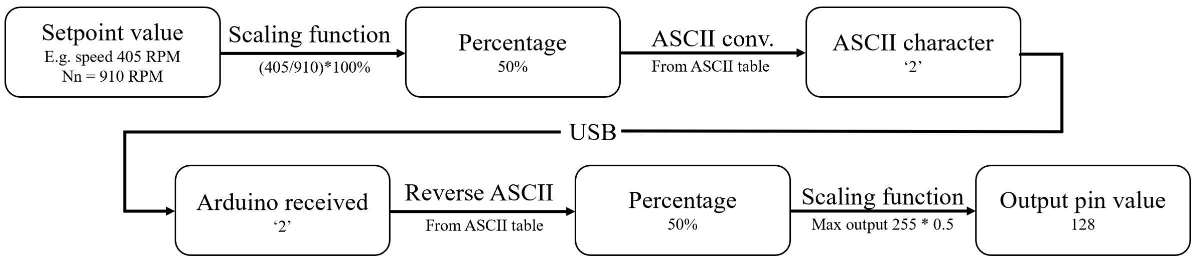


Figure 4.23: Example of a 50% speed setpoint value conversion from Python to Arduino

4.3.2 Arduino

The Arduino microcontroller described in subsection 4.2.4 comes as a blank microcontroller with no functions implemented when purchased. Therefore, all procedures to be performed by the Arduino must be programmed using the Arduino Programming Language, for example, creating scripts called sketches using the *Arduino IDE*. The sketches are written as plain text in the text editor and saved with the file extension `.ino`. As the Arduino is, in this setup, not supposed to contain any comprehensive control functions but merely act as a translator between the Python script and the equipment, its main parts revolve around receiving, sorting and performing commands received through the serial bus.

The sketches' structure can vary to a certain extent, although two main void functions, `void setup()` and `void loop()`, always have to be included. The setup function runs once every time the sketch is reloaded to the Arduino and is followed by a continuously repeating loop function. Of course, one can further extend the sketch by creating additional customised functions, but these will only execute if called for in the setup or loop function. Arduino uses the communication function `Serial` to communicate with a computer or another device. This section further describes the complete Arduino sketch found in Appendix C

Decode serial messages. The main `loop()` is kept short and transparent by creating separate functions for each functionality described in Table 4.3. As illustrated in Figure 4.24 the `loop()` begins with an if-statement checking the serial bus for newly available data every iteration. If no new data is available, the Arduino will continue running in its previous state without changing any parameters. On the other hand, if data *is* available, the sketch runs the function named `recvWithEndMarker()`, which reads the received data, stores it in an array until it gets the end character and then, if the data was collected successfully, calls a function named `parseInput()`. `parseInput()` splits the received array using x as separation and essentially decides which of the main functions to run. If the function receives multiple characters and the second is not an `x`, there is something wrong with the string. The sketch will then return to the main loop and do nothing except ensure the variable notifying the arrival of new data, `newData`, is set to `False`.

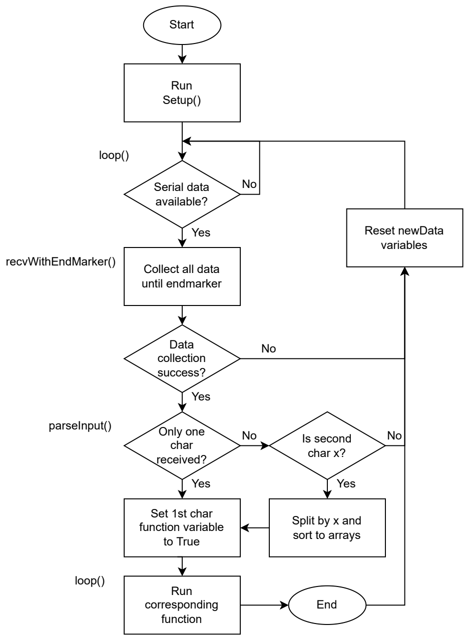


Figure 4.24: Flowchart of how Arduino selects which function in Table 4.3 to run

Read analogue values. The single-character serial commands control functions where there is no need to specify which channel the communication is directed to, such as reading analogue inputs and switching the internal LED on/off. While the LED switch is not relevant for controlling the I/O, it comes handy during communication troubleshooting. However, the read analogue input function serves a more vital purpose as it records the feedback from the physical devices. The function runs a for loop to collect the voltage levels, represented as a number between 0 and 2^{10}, stores them in an array and transmits that array to the serial bus.

One thing to notice is that the Arduino does not have the same speed issues when sending data to the serial bus as when receiving data, which means the data transmitted from the Arduino can be sent as raw multi-digit integers. This means they do not have to be converted into ASCII characters as the values sent to the Arduino have to (see Figure 4.23).

Write to analogue and digital outputs. When the function `parseIntput()`, as shown second to last in Figure 4.24, splits the received array into three different values, these values are stored in three global variables. One describes the channel type, one represents the channel number, and one declares the new channel value. These are stored globally as they are overwritten when a new command arrives, and it allows access to all functions without running the parsing function multiple times, as that quickly slows down the Arduino. As shown in Figure 4.25, the first two variables, pictured as `a` and `b` in Figure 4.23, are run through a series of if- statements and switch cases to determine which channel is to receive a new setpoint, the final value `c`. If the channel is an analogue output, the c value is scaled to its corresponding value (as described in Figure 4.23) and then written to the analogue PWM output pin. If the channel is digital, the digital out pin is set to `HIGH` if the value is equal to or greater than 50.

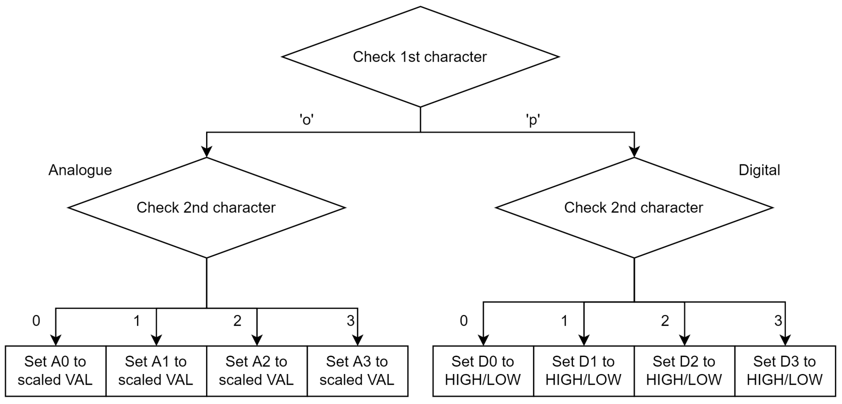


Figure 4.25: Select output Arduino

When an analogue output changes setpoint (SP_{recieved}), its output values ($OutVal_{\textbf{Arduino}}$) are scaled according to Figure 4.23 converted to Equation (4.2), where Max_{output} is the maximum possible Arduino output value, usually 255 on 8-bit PWM outputs, unless adjusted by the *calibration* function:

$$OutVal_{\textbf{Arduino}} = \left(\frac{SP_{\text{received}}}{100}\right) \cdot Max_{\text{output}} \tag{4.2}$$

Calibration. The functions described in paragraph 4.3.2 and converters described in subsubsection 4.2.5 are based on an Arduino PWM output voltage ranging from $0-5V$. However, some Arduinos tend to output a maximum voltage higher than 5V when writing 255 to a PWM output. Therefore, the calibration function is created to combat the slight overvoltage by adjusting the maximum output value down from 255 to a value that results in a more desired output voltage. The function is based on the output voltages *after* the converters, i.e. $10V$ being the desired top value, and uses the same structure to receive values to assigned channels as described in subsection 4.3.1. The first character signals the calibration mode to start, the second one signals the channel number, and the third declares the measured voltage level as a number between 0 and 100, representing the voltage between $10V$ and $11V$. If the measured voltage is outside this range, one must take other measures, such as adjusting the converters' resistance as described in subsubsection 4.2.5. The function calculates the new maximum output using Equation (4.3), where Max_{output} is the previous maximum output value and val_{received} is the value representing a voltage between $10V$ and $11V$

$$Max_{\text{newOutput}} = \left(\frac{URV - LRV}{\frac{(URV-LRV)+(\frac{val_{\text{received}}}{100})}{Max_{\text{output}}}}\right) \tag{4.3}$$

The new maximum value for each channel is stored in a global variable and used to scale every forthcoming analogue output. The function can also reset the maximum value to its original 255 by receiving the character '`r`' instead of the measured value.

4.3.3 Python

While the Arduino acts as the bridge between the virtual environment and physical, analogue values, it contains no form of controller. Instead, as illustrated in Figure 4.21, the Python script is responsible for roles such as user interaction, data scaling, data presentation and other controller functions. Python is an open-source, object-oriented programming language widely used due to its accessibility, extensive libraries and large community, and chosen as the primary language for this application for the same reasons [62].

The script is based around the package named Pyserial for serial communication with the Arduino through USB. The Pyserial package facilitates serial communication, allowing data exchange between the python script and external hardware through, for example, USB. Pyserial has to be installed and imported to the script as a package `import serial`. [63] The line `ser = serial.Serial('COM3', baud rate = 9600, timeout = 1)`, is used to open the serial port. This example is taken directly from the script found in Appendix B where `'COM3'` corresponds to the port used by Arduino, the baud rate of `9600` matches the Arduino Uno's baud rate and the timeout is set to 1s. `ser.write()` sends data to the serial bus using this line, while `ser.read()` reads data from the serial bus in a similar fashion. To avoid the Arduino timeout-issue described in subsection 4.2.4, all data from Python is converted to individual bytes before being sent to the bus.

User Interaction. The python script, found in Appendix B consists of multiple functions serving various purposes. For example, the application() function, the main function running when compiled, comprises a text-based user interface continuously asking the user for input. Depending on the input, the script chooses which actions to perform and what commands are sent to the Arduino. The script consists of five main processes, listed in Table 4.4.

Table 4.4: Main commands in Python script

Command	Action
read	Read from analogue inputs
write	Write to analogue/digital outputs
cal	Initiate calibration sequence
on/off	Switch internal LED on or off
q	Exit main loop and close Arduino's serial port

The user input runs through a function [64] checking if the input value is defined as valid. If anything *not* specified in Table 4.4 is typed, the input function runs continuously until a valid input is inserted. Then, depending on the input, the script either runs a corresponding function directly or asks for follow-up input.

Read analogue inputs. Reading the feedback values from the VFD and servo drive requires the python script to send a request to the Arduino, asking it to print the recorded values onto the serial bus. As declared in Table 4.3, this request is in the form of an `'e'` sent to the bus. When the Arduino receives the `'e'`, it runs through the function for recording its analogue input values and sends them back to the serial bus, where the

Python script reads the response. The function named `AI_read` is responsible for reading the Arduino's response and splitting the received string into its three original elements, while the `sortData()` makes sure to sort the received array into separate channels and store their values in different arrays. The process is repeated until the desired number of values, `numPoints`, are recorded, as illustrated in Figure 4.26.

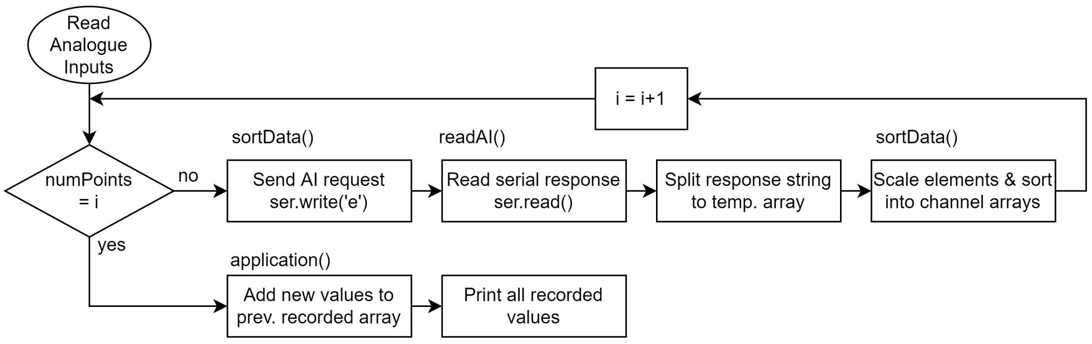

Read Analogue Inputs
i = i+1
sortData()
readAI()
sortData()
numPoints = i
no
Send AI request ser.write('e')
Read serial response ser.read()
Split response string to temp. array
Scale elements & sort into channel arrays
yes
application()
Add new values to prev. recorded array
Print all recorded values

Figure 4.26: Flowchart for reading analogue values in Python

Write to analogue and digital outputs. When writing to the analogue or digital outputs, the user is asked for three statements to be sent to the Arduino, as illustrated in Figure 4.27. These are if the output is an analogue or digital channel, its channel number and the desired output value. The response is sent to a function named `byteWriteArray()` illustrated below, which ends with printing the three variables to the serial bus.

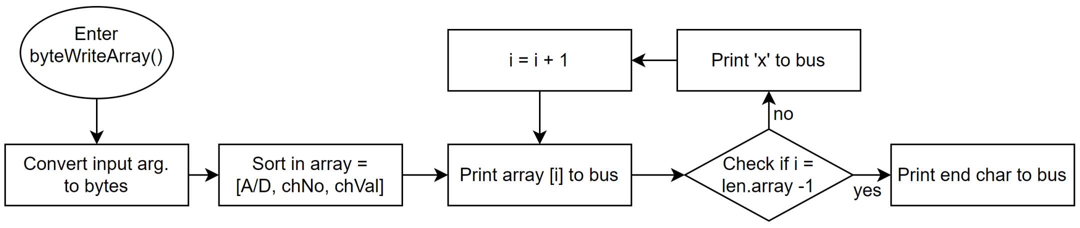


Figure 4.27: Flowchart for writing output values in Python

The function `byteWriteArray(AD,chNo,chVal)` takes its three input arguments respresenting analogue/digital (`AD`), channel number(`chNo`) and value(`chVal`), converts them from characters in string-format to byte values using the function `bytes(z, 'utf-8')` and stores the bytes in a temporary array. `'utf-8'`, short for *8-bit Unicode Transformation Format*, is one of the most popular character encoding methods used today, and what Arduino is able to read from the serial bus [65]. Next, the array of bytes is sent through a for loop, where each element is printed to the serial bus, separated by an `x` until its last element, which is followed by the character signifying the end of the message. This sequence results in a message perceived by the Arduino as the sequence illustrated in Figure 4.22.

Calibration. As described in paragraph 4.3.2, the Arduino sometimes has a higher PWM voltage output than its standard 5V, causing the maximum output voltage to be more than the expected $10V$. As a solution, the maximum analogue write value in the Arduino is adjusted down from 255 to a level that results in a $5V$ PWM output, based on the measured voltage after the converters. The Arduino receives the voltage level between $10V$ and $11V$, represented as a number between $0-100$. The python program's task is to record and send this value safely and efficiently.

During this process, the channels are set to their maximum output value, and the output voltages are measured using a voltmeter and typed into the script. As the calibration results in changed output values, the user is first asked to make sure all necessary equipment is disconnected or switched off before going through the analogue output channels one by one.

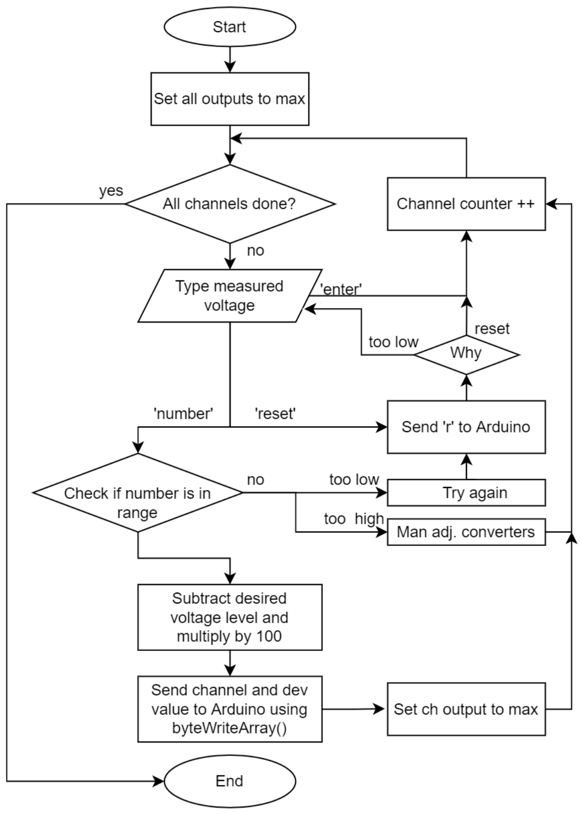


Figure 4.28: Flowchart for calibration sequence in Python

As illustrated in Figure 4.28, the sequence begins with setting all the analogue output channels to their maximum value. It then runs through a for-loop with one iteration for each channel, starting with channel AO_0, where the user is asked to measure the channel's output value and type it into the script. Here, the script accepts three different types of inputs:

- *Press 'enter'.* If the user presses 'enter' without any other information, the loop will skip to the next iteration (next channel) without any further action.
- *Type* `'reset'`. If the user types 'reset', the script sends an 'r' to the Arduino, causing the maximum output value to reset to 255.
- *Type a number, e.g.* `'10.25'`. If the user types a number, the script will check if it is within the specified range and then continue processing it.
- Any other data will not be accepted as input.

If the user inserts a number, the script checks if it is within the specified range, which in this case is between $10V$ and $11V$. Any voltage level above 11V is unlikely singularly caused by the Arduino, and the source of trouble is more likely to be found in the converters calibration. The same goes for any values below $9V$. If the voltage is between $-10V$, Arduino resets the maximum voltage, and the iteration is repeated. When the user inputs a voltage between $10-11V$, the value is scaled according to Equation (4.4), where LRV is $10V$, URV is $11V$, MV is the measured voltage and val_{sent} is the final value transmitted to the Arduino.

$$\text{val}_{\text{sent}} = \frac{MV - LRV}{(URV - LRV)} \cdot 100 \tag{4.4}$$

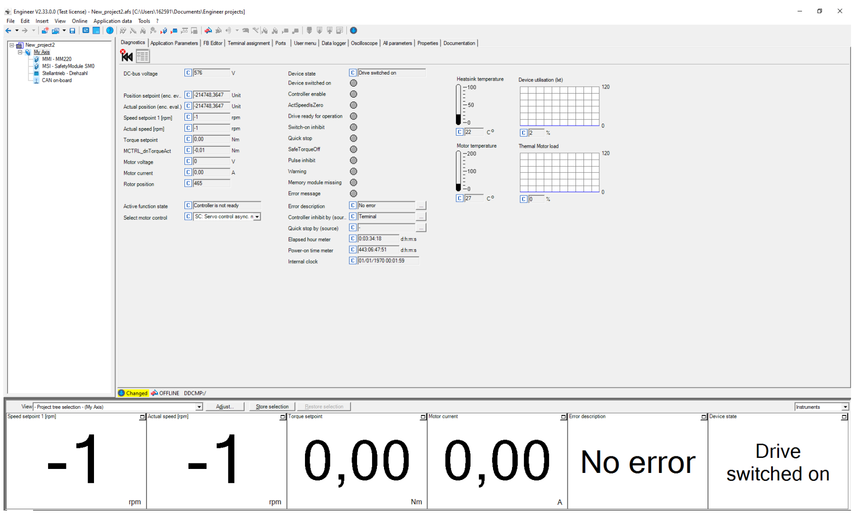
Scaling. As the number exchanges from Python to Arduino are all represented as values between 0-100, the script also contains a beneficial yet straightforward scaling function. The function takes five input arguments: The input valueinValue, its range represented by the max and min value in_{min} and in_{max} and the output range out_{min} and out_{max}. It returns a single output number scaled according to Equation (4.5), bearing quite a few resemblances to the scaling of analogue values presented in Equation (3.3).

$$\text{inValue} = \frac{\text{inValue} - \text{in}_{\text{min}}}{\text{in}_{\text{max}} - \text{in}_{\text{min}}} \cdot (\text{out}_{\text{max}} - \text{out}_{\text{min}}) + \text{out}_{\text{min}} \tag{4.5}$$

4.3.4 Lenze Engineer

As described in subsection 3.2.1, the servo drive in the original test stand is controlled using the CANopen communication between the front panel and the drive. The majority of the analogue I/O terminals are not in use. Therefore, the software Lenze Engineer, a part of the EASY engineering tools package, comes into play to configure the drive, whether the I/O module setup or other application aspects. Engineer is an extensive software with a significant number of functionalities, and as a result, this section will only cover the basics needed to configure the I/O terminals.

When initialising the software, one gets the option of creating a brand new project, opening an existing project or uploading data from an online device. As the goal is to add the I/O as an extension of the current control interfaces and *not* changing the functionalities already implemented in the drive, uploading data from the device is wise. Connecting the servo drive to the computer using the diagnostics adapter pictured in Figure 4.2b and pressing *"upload data from system"* imports all the device settings, functionalities and other parameters *from* the servo drive *to* the Engineer software.


File Edit Insert View Online Application data Tools ?
New_project2
My Axis
MMI - MM220
MSI - SafetyModule SM0
Stellantrieb - Drehzahl
CAN on-board
Diagnostics | Application Parameters | FB Editor | Terminal assignment | Ports | User menu | Data logger | Oscilloscope | All parameters | Properties | Documentation
Changed OFFLINE DDCMP:/
Adjust...
Store selection
Restore selection
Instruments

Engineer V2.33.0.0 (Test license) - New_project2.afs [C:\Users\162591\Documents\Engineer projects]

DC-bus voltage	576	V
Position setpoint (enc. ev..	-214748,3647	Unit
Actual position (enc. eval.)	-214748,3647	Unit
Speed setpoint 1 [rpm]	-1	rpm
Actual speed [rpm]	-1	rpm
Torque setpoint	0.00	Nm
MCTRL_dnTorqueAct	-0.01	Nm
Motor voltage	0	V
Motor current	0.00	A
Rotor position	465	
Active function state	Controller is not ready	
Select motor control	SC: Servo control async. n	

Device state	Drive switched on	
Device switched on	●	
Controller enable	●	
ActSpeedIsZero	●	
Drive ready for operation	●	
Switch-on inhibit	●	
Quick stop	●	
SafeTorqueOff	●	
Pulse inhibit	●	
Warning	●	
Memory module missing	●	
Error message	●	
Error description	No error	
Controller inhibit by (sour..	Terminal	
Quick stop by (source)	-	
Elapsed hour meter	0:03:34:18	d:h:m:s
Power-on time meter	443:06:47:51	d:h:m:s
Internal clock	01/01/1970 00:01:59	

Heatsink temperature	22	C°
Motor temperature	27	C°
Device utilisation (Ixt)	2	%
Thermal Motor load	0	%

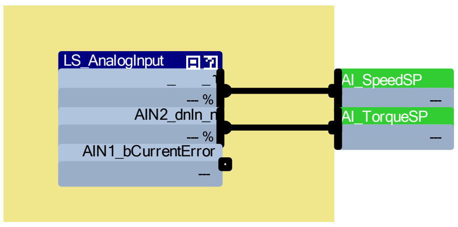
View - Project tree selection - (My Axis)

Speed setpoint 1 [rpm]	Actual speed [rpm]	Torque setpoint	Motor current	Error description	Device state
-1 rpm	-1 rpm	0,00 Nm	0,00 A	No error	Drive switched on

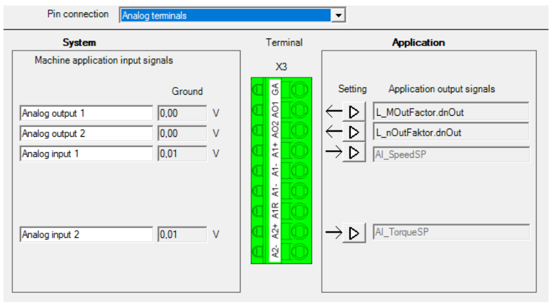
Figure 4.29: Lenze Engineer main window(device offline).

Once opened, the main Engineer window consists, as in Figure 4.29, of the main overview displaying the live Speed, torque, current consumption and other vital values. In the top mid tabs, one can navigate various parameters, editors, logs, etc. The relevant ones for adding the I/Os are the *Terminal assignment* tab and *FB editor*. The Terminal assignment gives an overview of the terminals' current tasks but has minimal editing options. To edit the terminal assignments, one must use the Function Block editor located in the tab next right. This tab contains the heart of the servo drive and essentially decides how the drive is to respond to various events and parameter changes.The complete Function Block Diagram imported from the servo drive is presented in Appendix E.

By adding an input block for the analogue input terminals and assigning their values to variables, `AI_speedSP` and `AI_torqueSP`, as shown in Figure 4.30a, the analogue inputs are added throughout the function block diagram to replace or add further control options to the already existent. The variables will then show up in the Terminal Assignment tab, connected to their corresponding analogue input as in Figure 4.30b.



(a) Analogue inputs in FB.



Pin connection: Analog terminals

System

Machine application input signals

	Ground	
Analog output 1	0,00	V
Analog output 2	0,00	V
Analog input 1	0,01	V
Analog input 2	0,01	V

Terminal

X3

GA AO1 AO2 A1+ A1- A1- A1R A2+ A2-

Application

Setting	Application output signals
← ▷	L_MOutFactor.dnOut
← ▷	L_nOutFaktor.dnOut
→ ▷	AI_SpeedSP
→ ▷	AI_TorqueSP

(b) Analogue Terminal Assignment tab.

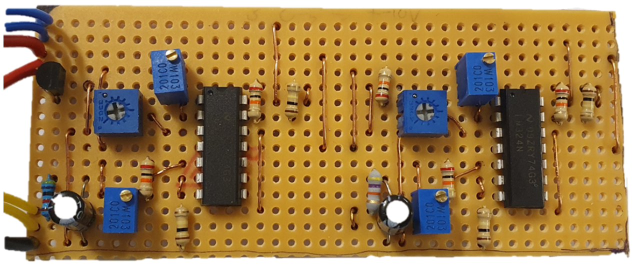
Figure 4.30: Analogue terminals in Lenze Engineer.

5 Analogue results

This chapter presents the results from the cases described in Chapter 4 and Section 4.3. Although the project revolves around creating a communication module rather than case-based studies, this chapter contains results based on the actual physical build and programming.

5.1 Converters

As presented in Chapter 4, the test stand's limited options for serial communication led to an I/O-based approach. Arduino was chosen as the most suitable link between the Python script and the test setup due to its high versatility, open-source protocol and budget-friendly cost. The converters required to adjust the voltage levels between the test stand and the Arduino were built according to subsubsection 4.2.5 and Appendix A and tested thoroughly to confirm their liability in various voltage ranges. The voltage tests' raw data are presented in Appendix A.


(a) $0-5V$ to $\pm 10V$.

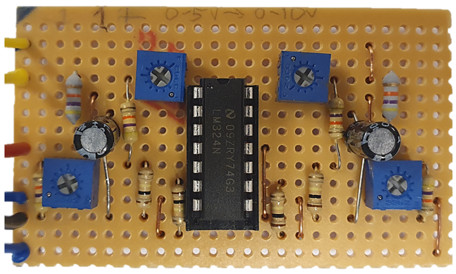

(b) $0-5V$ to $0-10V$.

Figure 5.1: Analogue output converters.

Figure 5.1 shown the two analogue voltage converters for amplifying the analogue PWM output value. Figure 5.1a shows the 0-5 to +-10V converter used for communicating with the Lenze drive, built according to the stripboard layout shown in Figure 4.14, apart from one main difference. The voltage regulator in channel two is swapped for a simple resistor acting as a voltage divider. This replacement is for one reason only: the lack of spare parts when it turned out one of the regulators was faulty.

Figure 5.1b shows the voltage doubler 0-5V to 0-10V built based on Figure 4.12 for communication with the VFD. Unfortunately, this circuit turned out to contain one significant weakness: The IC circuit does, for an unknown reason, start acting out when all its individual op-amps are used simultaneously. To avoid this issue, the circuit was extended, as shown in Figure 5.2, to include two ICs, where each IC only use two of the four available op-amps. The result are the circuit displayed in Figure 5.2 and Figure 5.3.

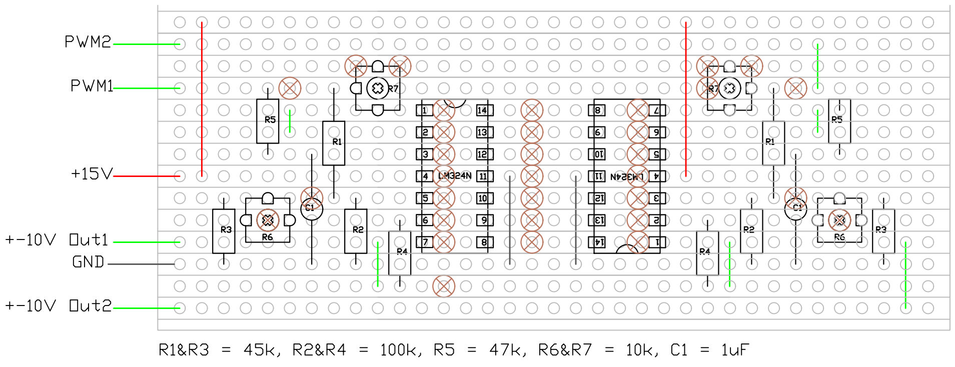


Figure 5.2: Revisited $0-5V$ to $0-10V$ converter stripboard.

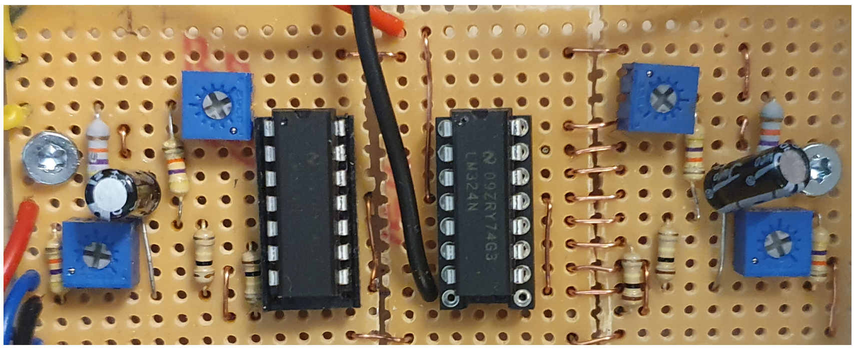


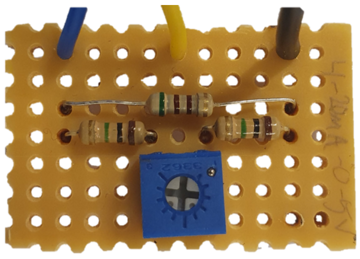
Figure 5.3: Revisited $0-5V$ to $0-10V$ converter build.

Figure 5.4 shows the circuits made for the analogue inputs, converting $4-20mA$ and $\pm 10V$ to microcontroller friendly voltage levels. Figure 5.4a is built based on Figure 4.18, and Figure 5.4b is made according to Figure 4.16.

The digital relay outputs described in Figure 4.20 were built as shown in Figure 5.5.

When all the converters, the power supply and the microcontroller are added together in a module, the result is a box displayed in Figure 5.6.

Pictures in Figure 5.7a and Figure 5.7b extracted from Appendix D show that the converters as well as the Arduino DAC have a linear response, providing correctly scaled voltages on all levels. In addition, Appendix D contains similar plots from all converters, which appear just as linear.


(a) $4-20mA$ to $0-5V$.

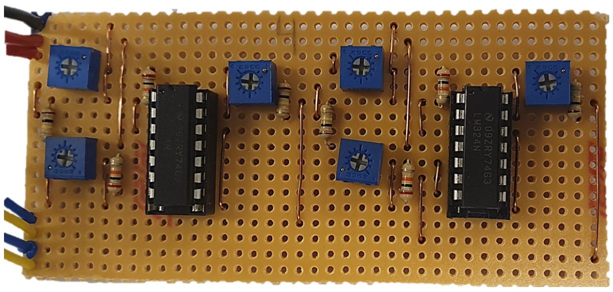

(b) $\pm10V$ to $0-5V$.

Figure 5.4: Analogue input converters.

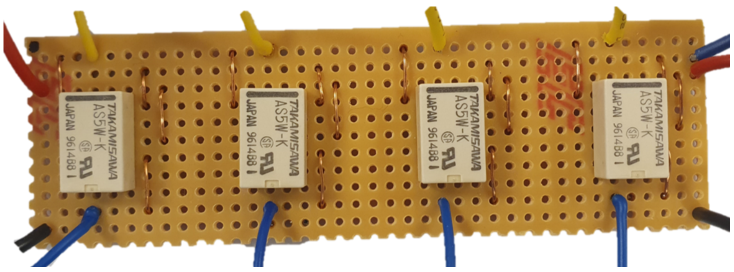


Figure 5.5: Digital outputs relay circuit

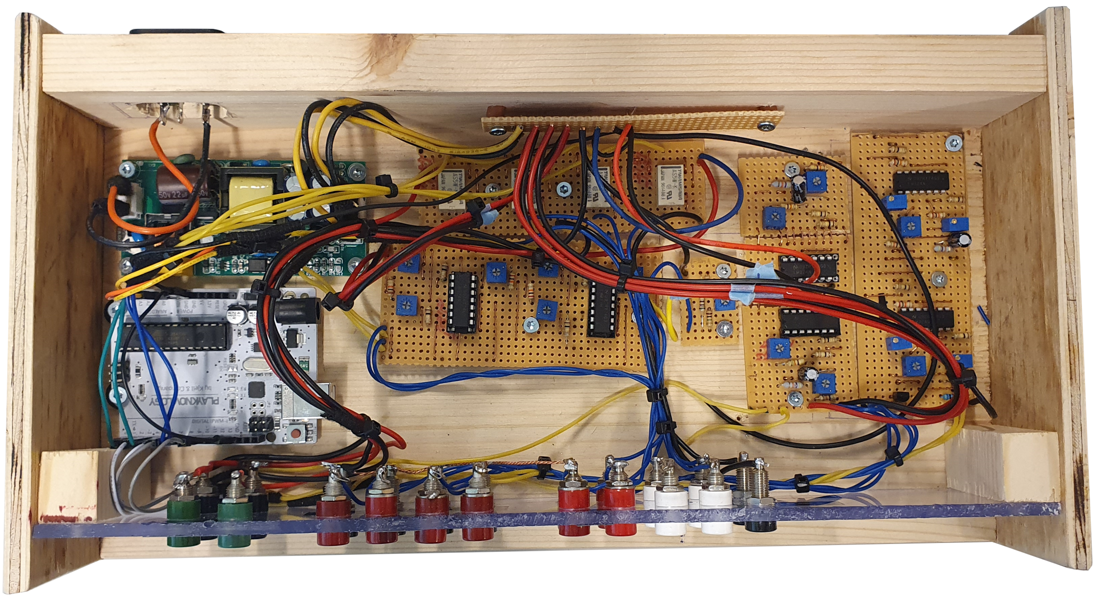
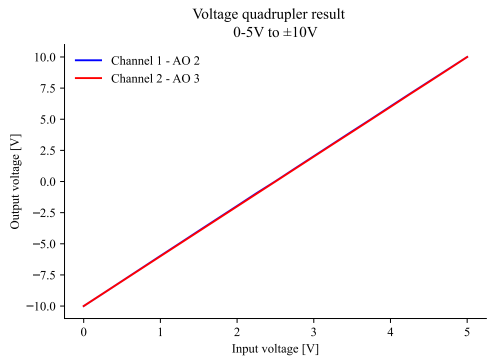


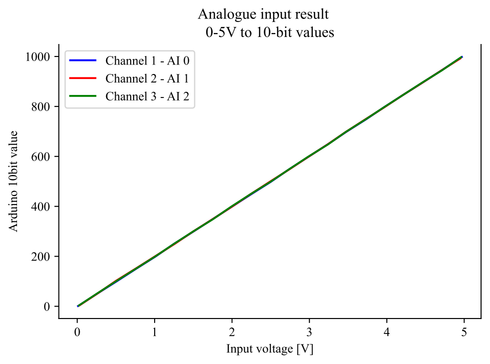
Figure 5.6: Complete I/O module.

5.2 Calibration

The Python script and Arduino sketch described in paragraph 4.3.3 and paragraph 4.3.2 are found in Appendix B Appendix C and contain a function for output calibration.



(a) $0-5V$ to $\pm 10V$ converter results.



(b) $0-5V$ to Arduino 10bit value.

Figure 5.7: Converter response.

Table 5.1: Calibration function results

	Pre calibration output		Post Calibration output	
Ch. no	**Arduino [V]**	**Channel [V]**	**Arduino [V]**	**Channel [V]**
AO 0	5.11	10.22	5.01	10.03
AO 1	5.09	10.20	5.00	10.01
AO 2	5.11	10.46	5.01	10.03
AO 3	5.11	10.22	5.01	10.03

Table 5.1 shows the results from running the calibration function. The results are relatively accurate and reach the proper levels instantaneously.

6 Discussion

An Arduino Uno was picked to act as the immediate translator between a computer and the test stand, as that allows for control using Python and pySerial. All control functions lie within the Python script, which transmits messages through USB that tells the Arduino to raise, lower or read its pin values. Then, converters are lined up between the Arduino pins and the test stand terminals to transpose the signal level to a safe, coherent level for the sending and receiving equipment.

subsection 3.3.1 describes the serial communication alternatives available on the servo drive. Although direct serial communication between a computer and the drive would have allowed for a more accessible, less wire-consuming communication, it does open the doorway to a few other pitfalls, such as incompatible converters, ineffective signal handling and clashing firmware updates. The more classic analogue I/O communication avoids in all its simplicity these pitfalls, although it is more susceptible to disconnected wires, inaccurate signal scaling and requires more floor space.

Section 4.2 goes through the various DAC modules considered the communication link, beginning with the National Instruments Compact DAQ. Using a DAC device allows for the benefits of digitally programmed control sequences, combined with the upsides of the non-brand-specific, effortlessly applicable analogue communication. The main downside is adding an extra, bulky device as the middle man. The NI compact DAQ is a reliable, customisable device with premade modules for correct signal levels and a Python library made easy to use, allowing for effortlessly implementation into a Python-based control system. The compact DAQ was initially the favoured choice of I/O-module for this setup if it had been more budget-friendly.

In searching for a more affordable solution, the microcontroller Arduino became the controller of choice due to its plenty of channels and the fully programmable response system connected to those pins. However, its lack of premade Python communication libraries makes it very customisable, albeit susceptible to programming errors. Fortunately, Arduino has a vast community, and due to its open-source mindset, there are plenty of forums and guidelines found online. The voltage converters needed between the signal and the Arduino take up space. However, they do not pose any other significant disadvantages to the setup, as they are all built with solid, mostly passive components prune to last.

A considerable disadvantage of using the Arduino as an intermedia between the analogue values and the Python scripts is its slow response time to string messages received through the serial port, creating the need subsection 4.3.1. As this system consists of sending multiple individual byte messages in a row until the end character is received, it is susceptible to a large number of plausible errors. It also reduces the resolution of the outgoing control signal, as every value has to be scaled to an integer between 0 and 100 to fit the ASCII conversion system created in subsection 4.3.1. Future work should include improving the transmission integrity and increasing the outgoing resolution.

As the entire control script is built using Python, increasing the use of the setup further should not pose any immediate unattainable challenges due to the wide range of API (Application Programming interfaces), libraries and functions created for it. The perhaps biggest challenge is, at this point, ensuring continuous data integrity when constantly transmitting and reading values through serial communication.

7 Conclusion

The servo drive test stand from Lucas-Nulle, its components and its software have been analysed and reviewed throughout this thesis. The stand consists of a servo drive and motor controlled using CANopen and a locally controlled test motor and frequency drive. The stand's provided software ActiverServo allows for, through USB communication, basic load emulations such as pumps, hoist drives and constant setpoints but comes to short if one wishes to implement it into other systems. To extend the test stand's control systems, one should bypass the existing control interface and access the drive directly through CANopen, its diagnostics port or I/O terminals. The diagnostics port allows for altering the drive parameters but not direct control, and the CANopen port is restricted to a brand-specific adapter outside this project's budget, leaving control through the I/O terminals the only viable option.

The low-cost I/O module created to fit the stand's needs is based on an Ardunio and voltage converters explicitly built to fit the conversion between the Arduino's pins and the servo drive and frequency drives I/O-terminals. The Arduino merely acts as a translator between a Python script and the analogue values, and the Python script is responsible for all control commands. As the control is Python-based, it allows for future comprehensive extensions, including entire simulation models, either programmed in Python or through functional mock-up units.

The project has resulted in a solid base for future extensions. Future work on the setup may include, for example, adding a graphical user interface, creating a storage system for saved data, improving the data security in the Python Arduino communication and, last but not least, creating a system for further extensions to other simulation models.

Bibliography

[1] J. P. B. a. C. K. Ebinger, *The Electricity Revolution*, en-US, Nov. 2001. [Online]. Available: https://www.brookings.edu/research/the-electricity-revolution/ (visited on 16/05/2022).

[2] Julian Critchlow, *What is the future of electricity?* en, Feb. 2015. [Online]. Available: https://www.weforum.org/agenda/2015/04/what-is-the-future-of-electricity/ (visited on 16/05/2022).

[3] N. G. Society, *Renewable Energy*, en, Feb. 2013. [Online]. Available: http://www.nationalgeographic.org/article/renewable-energy/ (visited on 16/05/2022).

[4] Amy Bennett, *Electric Car rEVolution*, en-GB, Nov. 2021. [Online]. Available: https://carbonliteracy.com/electric-car-revolution/ (visited on 16/05/2022).

[5] E. Wiik, *F Faktura Ordrenr 2450*, Norsk, Nov. 2012. (visited on 14/02/2022).

[6] D. Winkler, *Task Description: Development of an open control interface for a servo machine test stand*, Jan. 2022.

[7] *Lucas Nülle - Servo machine test stand for 1kW machines incl. software ActiveServo (D,GB,F,E)*. [Online]. Available: https://www.lucas-nuelle.com/1004/pid/26351/apg/13659/Servo-machine-test-stand-for-1kW-machines-incl-software-ActiveServo-D,GB,F,E-.htm (visited on 24/04/2022).

[8] S. Bharadwaj, *Advantages & Disadvantages Induction Motor*, en-US, Apr. 2016. [Online]. Available: https://instrumentationtools.com/advantages-disadvantages-induction-motor/ (visited on 28/01/2022).

[9] A. Princy, *Induction Motors: Main Types and Different Applications*, Apr. 2020. [Online]. Available: https://www.researchdive.com/blog/induction-motors-main-types-and-different-applications (visited on 30/01/2022).

[10] S. Mathew, *How does an induction motor work?* 2019. [Online]. Available: https://www.lesics.com/how-does-an-induction-motor-work.html (visited on 30/01/2022).

[11] C. Globe, *Construction of Induction Motor*, en-US, Jan. 2016. [Online]. Available: https://circuitglobe.com/construction-of-induction-motor.html (visited on 04/02/2022).

[12] O. PLanas, *Squirrel Cage Rotor | Asynchronous or Induction Motor, Characteristics and Operation*, Oct. 2018. [Online]. Available: `https://en.demotor.net/electric-motors/ac-motors/asynchronous-motor/squirrel-cage-rotor` (visited on 04/02/2022).

[13] *Squirrel Cage Induction Motor: Working Principle & Applications*, Aug. 2020. [Online]. Available: `https://www.electrical4u.com/squirrel-cage-induction-motor/` (visited on 06/02/2022).

[14] C. Globe, *Starting of a Synchronous Motor - Prime Mover & Damper Winding*, en-US, Jan. 2016. [Online]. Available: `https://circuitglobe.com/starting-of-synchronous-motor.html` (visited on 20/04/2022).

[15] E. Deck, *What is Slip in Induction Motor? - Effect of Slip on Induction Motor*, en, Nov. 2020. [Online]. Available: `https://www.electricaldeck.com/2020/11/slip-in-induction-motor-and-effect-of-slip-on-induction-motor.html` (visited on 20/04/2022).

[16] Sushmita, *Electrical Braking in Polyphase Induction Motors*, en-US, Feb. 2018. [Online]. Available: `https://www.engineeringenotes.com/electrical-engineering/electric-braking/electrical-braking-in-polyphase-induction-motors-electrical-engineering/37426` (visited on 21/04/2022).

[17] A. Knight, *Electrical Machines - Induction Motor Torque Speed Curve*, English, Educational, Sep. 2018. [Online]. Available: `https://people.ucalgary.ca/~aknigh/electrical_machines/induction/im_trq_speed.html` (visited on 21/04/2022).

[18] Danfoss, *What is a variable frequency drive? | Danfoss*, Apr. 2022. [Online]. Available: `https://www.danfoss.com/en/about-danfoss/our-businesses/drives/what-is-a-variable-frequency-drive/` (visited on 21/04/2022).

[19] S. U. Hassan and H. B. Akram, 'Speed and Frequency Control of AC Induction Motor Using Variable Frequency Drive,' en, *Department of Electrical Engineering, Institute of Space Technology - Student Research Paper Conference*, vol. 2, p. 8, 2015.

[20] ATO, *AC Drives Basics (Benefits, Principle and Theory)*, Jul. 2017. [Online]. Available: `http://www.acdrive.org/ac-drives-basics.html` (visited on 22/04/2022).

[21] M. Malinowski, M. P. Kazmierkowski and A. M. Trzynadlowski, 'A comparative study of control techniques for PWM rectifiers in AC adjustable speed drives,' *IEEE Transactions on Power Electronics*, vol. 18, no. 6, pp. 1390–1396, Nov. 2003, ISSN: 1941-0107. DOI: `10.1109/TPEL.2003.818871`.

[22] T. Bellei, R. O'Leary and E. Camm, 'Evaluating capacitor-switching devices for preventing nuisance tripping of adjustable-speed drives due to voltage magnification,' *IEEE Transactions on Power Delivery*, vol. 11, no. 3, pp. 1373–1378, 1996. DOI: `10.1109/61.517494`. [Online]. Available: `https://ieeexplore.ieee.org/document/517494` (visited on 22/04/2022).

[23] S.-H. Kim, 'Chapter 7 - Pulse width modulation inverters,' in *Electric Motor Control*, S.-H. Kim, Ed., Elsevier, Jan. 2017, pp. 265–340, ISBN: 978-0-12-812138-2. DOI: 10.1016/B978-0-12-812138-2.00007-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128121382000076.

[24] J. Rodriguez, J. Dixon, J. Espinoza, J. Pontt and P. Lezana, 'PWM regenerative rectifiers: State of the art,' *IEEE Transactions on Industrial Electronics*, vol. 52, no. 1, pp. 5–22, Feb. 2005, Conference Name: IEEE Transactions on Industrial Electronics, ISSN: 1557-9948. DOI: 10.1109/TIE.2004.841149.

[25] K. Sharma, '6 - Automation Strategies,' in *Overview of Industrial Process Automation*, K. Sharma, Ed., London: Elsevier, Jan. 2011, pp. 53–62, ISBN: 978-0-12-415779-8. DOI: 10.1016/B978-0-12-415779-8.00006-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124157798000061.

[26] *Rotasyn Standard Resolvers*, en-US. [Online]. Available: http://https%253A%252F%252Fwww.admotec.com%252Fresolver%252Fstandard-rotasyn-sensors%252F (visited on 24/04/2022).

[27] *Servomotors, Stepper Motors, and Actuators for Motion*, Oct. 2010. [Online]. Available: https://uniquemachines.blogspot.com/2010/10/servomotors-stepper-motorsand-actuators.html (visited on 23/04/2022).

[28] Lenze, *9400 HighLine servo inverter*, Sep. 2018. [Online]. Available: https://www.lenze.com/en-us/products/inverters/servo-inverters/9400-highline-servo-inverter/ (visited on 24/04/2022).

[29] *Lucas Nülle - Dynamic servo machine test system for 0.3kW machines incl. software ActiveServo*, May 2018. [Online]. Available: https://www.lucas-nuelle.us/2776/pid/22637/apg/11298/Dynamic-servo-machine-test-system-for-03kW-machines-incl-software-ActiveServo.htm (visited on 28/01/2022).

[30] B. Mehta and Y. Reddy, 'Chapter 9 - Serial communications,' in *Industrial Process Automation Systems*, B. Mehta and Y. Reddy, Eds., Oxford: Butterworth-Heinemann, Jan. 2015, pp. 307–339, ISBN: 978-0-12-800939-0. DOI: 10.1016/B978-0-12-800939-0.00009-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128009390000097.

[31] D. Ibrahim, 'Chapter 5 - Simple PIC18 Projects,' en, in *PIC Microcontroller Projects in C (Second Edition)*, D. Ibrahim, Ed., Oxford: Newnes, Jan. 2014, pp. 67–171, ISBN: 978-0-08-099924-1. DOI: 10.1016/B978-0-08-099924-1.00005-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080999241000058 (visited on 05/05/2022).

[32] CSS Electronics, *CAN Bus Explained - A Simple Intro [v2.0 | 2021]* , Oct. 2021. [Online]. Available: https://www.youtube.com/watch?v=oYps7vT708E (visited on 05/05/2022).

[33] M. Falch, *CAN Bus Explained - A Simple Intro [2022 | The #1 Tutorial]*, en, Apr. 2022. [Online]. Available: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial (visited on 05/05/2022).

[34] *Lenze L-Force 9400 Servo Drives Software Manual*, 2006.

[35] *USB 2.0 cable wiring pinout diagram @ pinoutguide.com*, Apr. 2022. [Online]. Available: https://pinoutguide.com/SerialPortsCables/usb_cable_pinout.shtml (visited on 06/05/2022).

[36] C. Peacock, *USB in a NutShell - Chapter 3 - USB Protocols*, Sep. 2010. [Online]. Available: https://beyondlogic.org/usbnutshell/usb3.shtml (visited on 06/05/2022).

[37] *About the USB Protocol, Common USB Bus Errors, and How to Troubleshoot Them*, en, Section: News, Jul. 2020. [Online]. Available: https://www.totalphase.com/blog/2020/07/about-the-usb-protocol-common-usb-bus-errors-and-how-to-troubleshoot-them/ (visited on 06/05/2022).

[38] *Communication Protocols : Basics and Types with Functionality*, en-US, Nov. 2013. [Online]. Available: https://www.elprocus.com/communication-protocols/ (visited on 06/05/2022).

[39] *Instrumentation Basics: 4 - 20mA and 3 - 15psi Control Signals ~ Learning Instrumentation And Control Engineering*, Oct. 2019. [Online]. Available: https://www.instrumentationtoolbox.com/2011/01/instrumentation-basics-control-signals.html (visited on 06/05/2022).

[40] Y. Lee, *Pulse-Width-Modulation Digital-to-Analog Converter vs. Stand-Alone DAC*, en, Feb. 2017. [Online]. Available: https://www.designnews.com/pulse-width-modulation-digital-analog-converter-vs-stand-alone-dac-0 (visited on 07/05/2022).

[41] Eduvance, *PSoC Lecture 5 PWM Basics*, Aug. 2014. [Online]. Available: https://www.youtube.com/watch?v=1kET-moJ_Qw (visited on 07/05/2022).

[42] R. Aswinth, *What is PWM: Pulse Width Modulation*, en, Feb. 2022. [Online]. Available: https://circuitdigest.com/tutorial/what-is-pwm-pulse-width-modulation (visited on 07/05/2022).

[43] *Low Pass Filter - Passive RC Filter Tutorial*, en, Aug. 2013. [Online]. Available: https://www.electronics-tutorials.ws/filter/filter_2.html (visited on 07/05/2022).

[44] J. Wägenbach, *CAN bus topology and bus termination*, en-US, May 2021. [Online]. Available: https://support.maxongroup.com/hc/en-us/articles/360009241840-CAN-bus-topology-and-bus-termination (visited on 08/05/2022).

[45] *EMF2177IB Montageanleitung*, Jul. 2010. [Online]. Available: https://download.lenze.com/TD/EMF2177IB__CAN%20PC%20adapter%20USB__v3-1__DE_EN_FR.pdf?msclkid=616102fece6111ec90469459060b2461 (visited on 08/05/2022).

[46] Lenze, *Reference manual E94AxHE Servo Drives 9400 HighLine (Firmware 01-37)*, Mar. 2015. [Online]. Available: https://download.lenze.com/TD/E94AxHE__Servo%20Drives%209400%20HighLine%20(Firmware%2001-37)__v1-6__EN.pdf.

[47] *ABB ACS350 User's Manual*, 2007. [Online]. Available: https://library.e.abb.com/public/2cf5b5aabb5777a9c125733d00407394/EN_ACS350%20UM_D.pdf.

[48] *Lenze Servo Drives 9400 Highline Reference Manual*, Apr. 2019. (visited on 09/05/2022).

[49] *iCP12 - usbStick (USB DAQ, PC Oscilloscope, Data Logger, Frequency Generator, PIC18F2550 IO Board)*, en, Aug. 2011. [Online]. Available: https://www.piccircuit.com/shop/pic-develop-board/119-160-icp12-usbstick-pic18f2550-io-board.html?msclkid=89f13523cf3711ecabc7128a25f84729 (visited on 09/05/2022).

[50] *Arduino - ArduinoBoardUno*, Apr. 2017. [Online]. Available: https://www.arduino.cc/en/Main/arduinoBoardUno&gt;?msclkid=047436d5cf3a11ec9773b7f04917bfe1 (visited on 09/05/2022).

[51] *What is Arduino?* en, May 2018. [Online]. Available: https://www.arduino.cc/en/Guide/Introduction (visited on 09/05/2022).

[52] *Playknowlogy Uno Rev. 3 Arduino-kompatibelt utviklingskort - Utviklingskort*, nb, Feb. 2022. [Online]. Available: https://www.kjell.com/no/produkter/elektro-og-verktoy/arduino/utviklingskort/playknowlogy-uno-rev.-3-arduino-kompatibelt-utviklingskort-p88860 (visited on 09/05/2022).

[53] *Online circuit simulator & schematic editor - CircuitLab*, Mar. 2020. [Online]. Available: https://www.circuitlab.com/ (visited on 11/05/2022).

[54] *AutoCAD_programvare*, no-NO. [Online]. Available: https://www.autodesk.no/products/autocad/overview (visited on 11/05/2022).

[55] *ACDC Switching Transformer Board*, en-US, Mar. 2014. [Online]. Available: https://www.ebay.com/itm/322743968466 (visited on 13/05/2022).

[56] STMicroelectronics, *LM324N datasheet*, 1999. [Online]. Available: https://pdf1.alldatasheet.com/datasheet-pdf/view/22756/STMICROELECTRONICS/LM324N.html (visited on 13/05/2022).

[57] *Operational Amplifier Basics, Types and Uses| Article | MPS*, Sep. 2019. [Online]. Available: https://www.monolithicpower.com/en/operational-amplifiers (visited on 13/05/2022).

[58] Kjetil Svendsen, *SV: Op-amp konvertering*, Mar. 2022. (visited on 16/03/2022).

[59] B. Carter, 'Chapter 2 - review of op amp basics,' in *Op amps for everyone (fourth edition)*, B. Carter, Ed., Fourth Edition, Boston: Newnes, 2013, pp. 7–17, ISBN: 978-0-12-391495-8. DOI: https://doi.org/10.1016/B978-0-12-391495-8.00002-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123914958000027.

[60] *Fujitsu A series miniature relay*, Sep. 2008. (visited on 09/05/2022).

[61] *ASCII Table - ASCII Character Codes, HTML, Octal, Hex, Decimal*, Apr. 2022. [Online]. Available: `https://www.asciitable.com/` (visited on 14/05/2022).

[62] Kayla MAtthews, *6 Reasons Why Python Is Suddenly Super Popular*, en-US, Section: 2017 Jul Opinions, Interviews, Jul. 2017. [Online]. Available: `https://www.kdnuggets.com/6-reasons-why-python-is-suddenly-super-popular.html/` (visited on 15/05/2022).

[63] *PySerial - Problem Solving with Python.* [Online]. Available: `https://problemsolvingwithpython` `com/11-Python-and-External-Hardware/11.01-PySerial/` (visited on 15/05/2022).

[64] Mateen Ulhaq, *Answer to "Asking the user for input until they give a valid response"*, Apr. 2014. [Online]. Available: `https://stackoverflow.com/a/23294659/18641112` (visited on 09/05/2022).

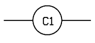
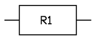
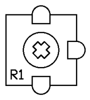
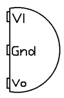
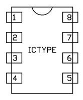
[65] J. Juviler, *What is UTF-8 Encoding? A Guide for Non-Programmers*, en, Oct. 2020. [Online]. Available: `https://blog.hubspot.com/website/what-is-utf-8` (visited on 16/05/2022).

Appendix A

Full resolution stripboard layouts forconverters

This paper contains the drawings created for the voltage converters stripboard layout. The drawings shows the normal, full layout front view in addition to the mirrored back-side, created to ease the placement of wire breaks. Symbol descriptions are found in Table A.1.

Table A.1: Stripboard circuits symbol description

Symbol	**Description**
🟩	Control signal wire
🟥	Power supply wire
—	Ground wire
⊗	Wire break
	Capacitor
	Resistor
	Potentiometer
	Voltage Regulator
	IC (size varies)

A.1 Tuning the converters

Instructions for finetuning the converters before first-time use.

Fine-tuning the $0-5V$ to $10V$ converter:

1. Set input voltage to $5V$ and adjust $R6$ to $10V$ out.
2. Use $R3$ for small adjustments.

Fine-tuning the $0-5V$ to $\pm 10V$ converter:

1. Switch $15V$ on and adjust $R7$ to $3V$ on pin 3.
2. Set input voltage to $0V$ and adjust $R6$ to $-10V$ out.
3. Set input voltage to $5V$ and adjust $R3$ to $10V$ out.
4. Repeat until stable.

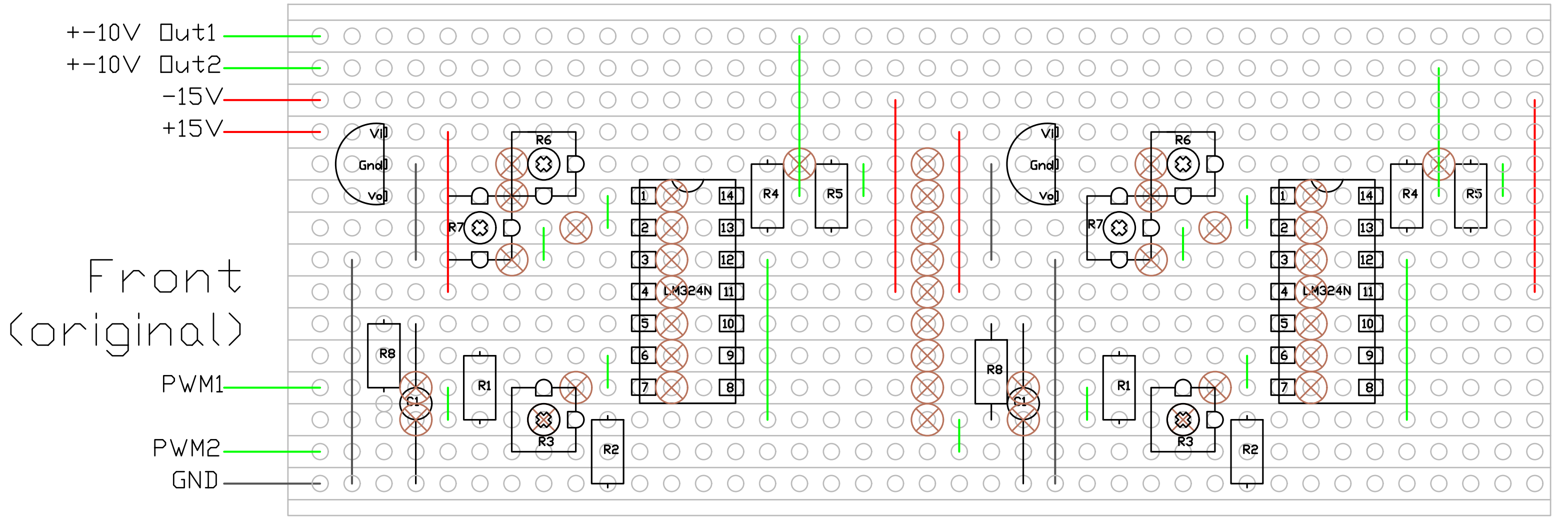
Fine-tuning the $\pm 10V$ to $0-5V$ converter:

1. Switch $-15V$ on and adjust $R7$ to $-10V$ on pin 1.
2. Set input voltage $\pm 10V$ to $-10V$ and adjust $R3$ to $0V$ out.
3. Set input voltage $\pm 10V$ to $+10V$ and adjust $R6$ to $5V$ out.
4. Repeat until stable.

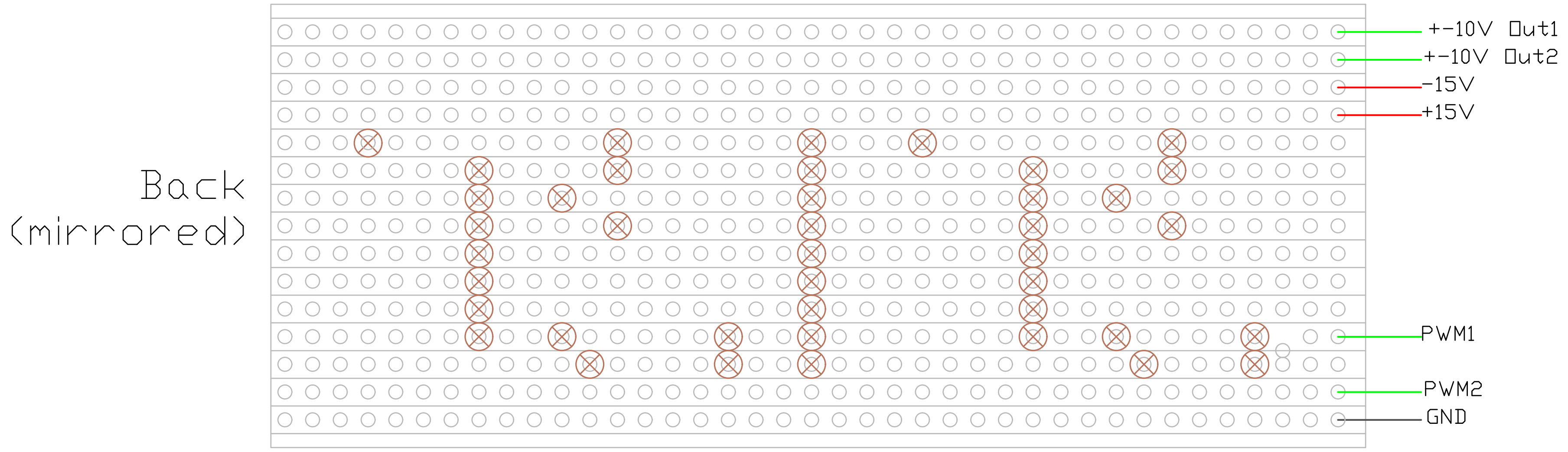
Fine-tuning the $4-20mA$ to $0-5V$ converter:

1. Set input current to $20mA$ and adjust potentiometer to $5V$ out.

RevNo	Revision note	Date	Signature	Checked



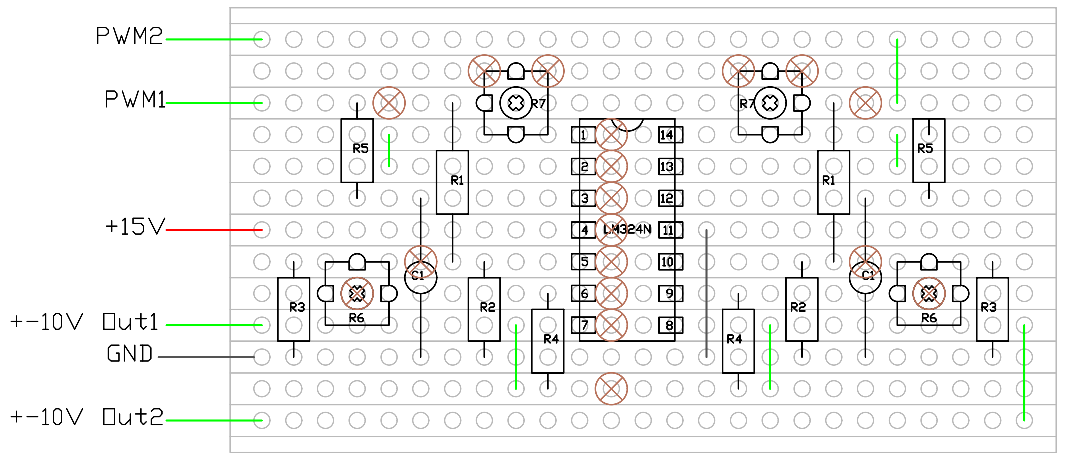
R1&R4 = 20k, R2&R5 = 100k, R3 = 10k normal R6&R7 = 10k slim, C1 = 1uF, Vr = LM78L05.
R6 and R7: Loose one leg. No ground. Use slim potentiometers.



Itemref	Quantity	Title/Name, designation, material, dimension etc			Article No./Reference	
Designed by ASK		Checked by	Approved by - date	Filename FVDoubler.dwg	Date 13.04.2022	Scale N/A
USN		0-5V to +-10V				
		05			Edition 1	Sheet 1/1

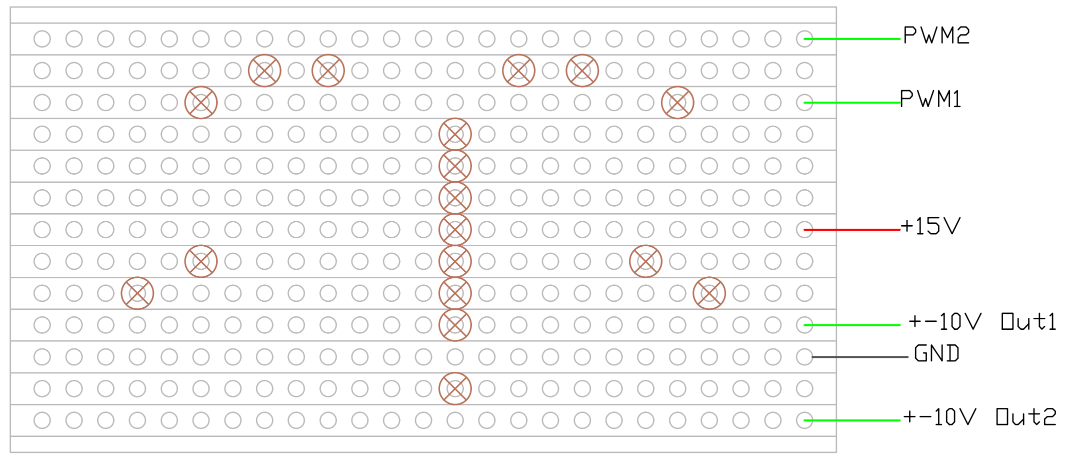
RevNo	Revision note	Date	Signature	Checked

Front (original)



R1&R3 = 45k, R2&R4 = 100k, R5 = 47k, R6&R7 = 10k, C1 = 1uF

Back (mirrored)

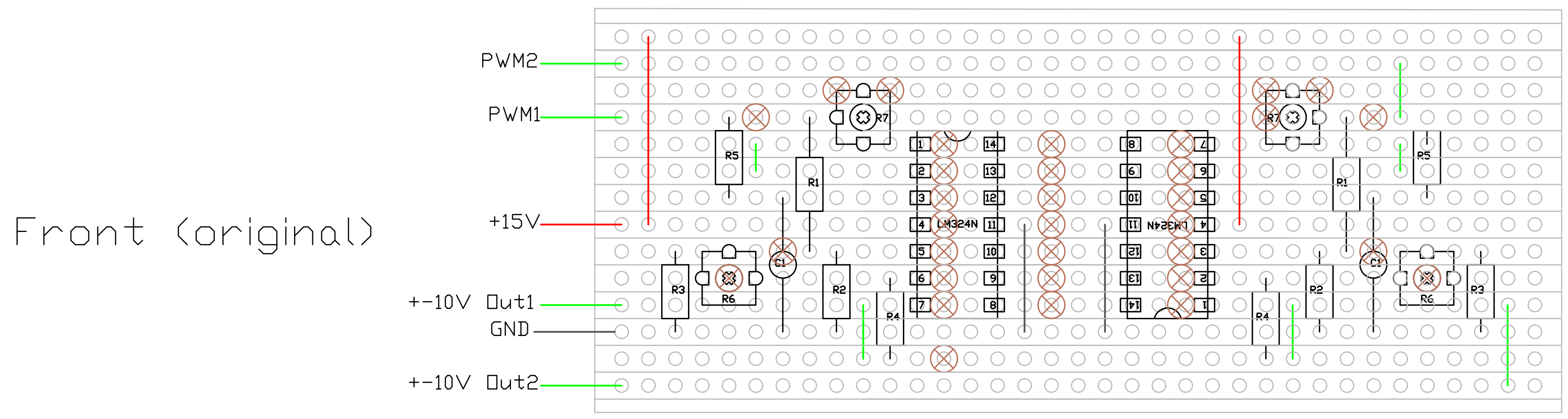


Itemref	Quantity	Title/Name, designation, material, dimension etc			Article No./Reference
Designed by ASK		Checked by	Approved by - date	Filename FVDoubler.dwg	Date 13.04.2022	Scale N/A

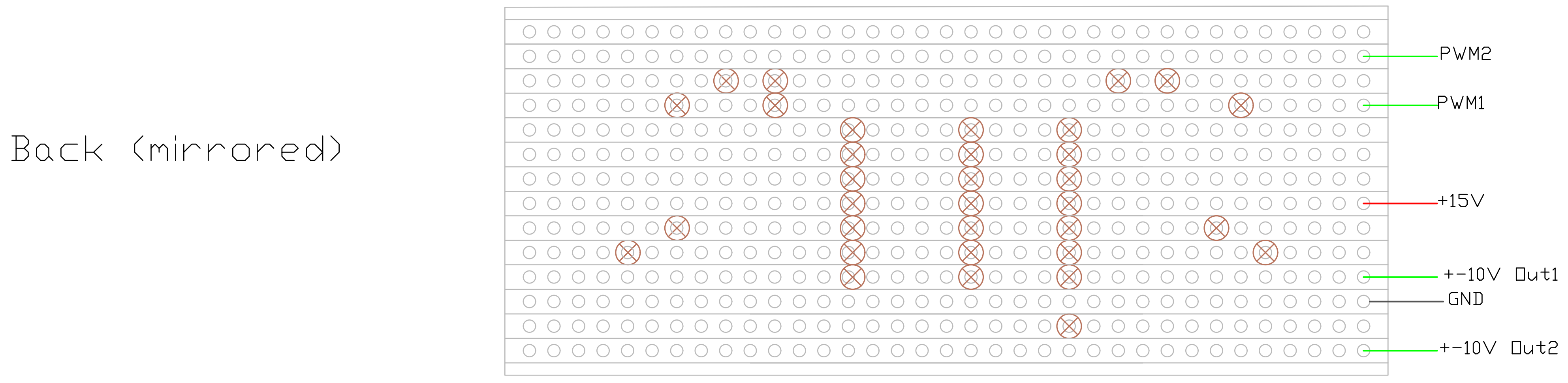
USN

0-5V to 0-10V

05	Edition 1	Sheet 1/1

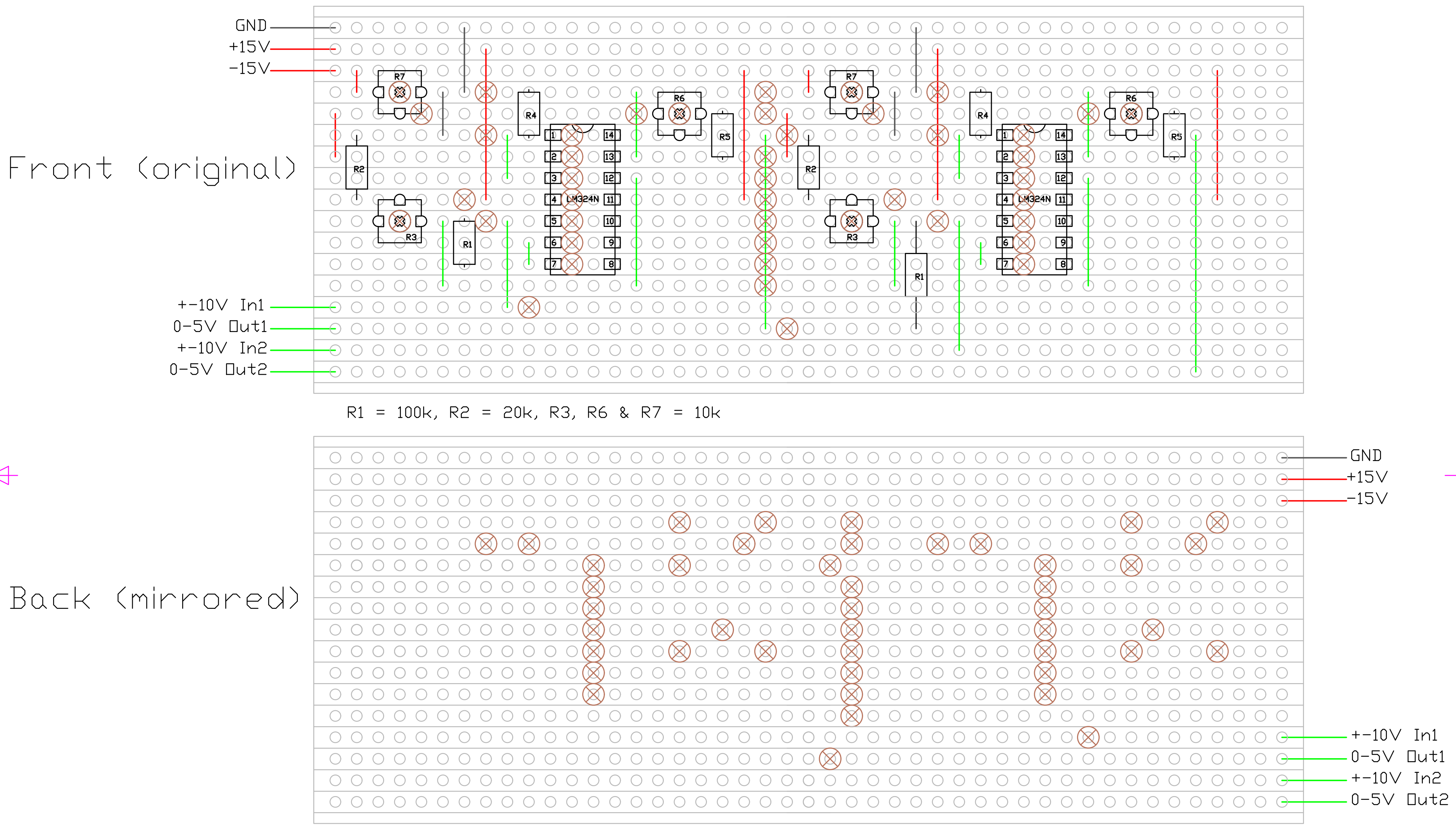


R1&R3 = 45k, R2&R4 = 100k, R5 = 47k, R6&R7 = 10k, C1 = 1uF



RevNo	Revision note	Date	Signature	Checked

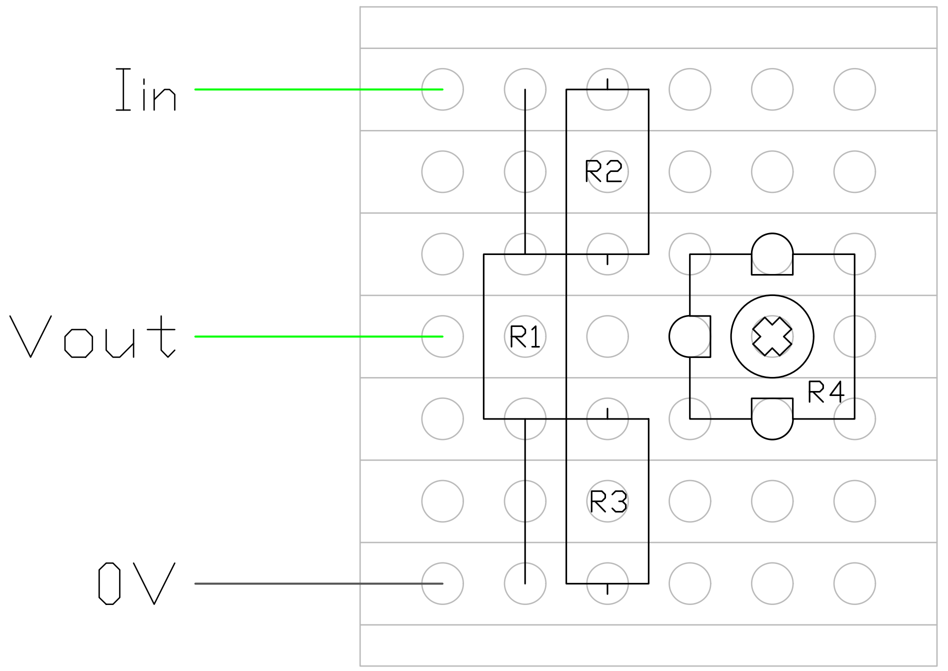
Itemref	Quantity	Title/Name, designation, material, dimension etc		Article No./Reference	
Designed by ASK	Checked by	Approved by - date	Filename FVDoubler.dgw	Date 18.05.2022	Scale N/A
USN		0-5V to 0-10V V2			
		05		Edition 2	Sheet 1/1



RevNo	Revision note	Date	Signature	Checked

Itemref	Quantity	Title/Name, designation, material, dimension etc		Article No./Reference	
Designed by ASK	Checked by	Approved by - date	Filename FVQuadr_reddwg	Date 13.04.2022	Scale N/A
USN		+-10V to 0-5V			
		05		Edition 1	Sheet 1/1

RevNo	Revision note	Date	Signature	Checked



R1 = 500R, R2 & R3 = 1M, R4 = 10k

Itemref	Quantity	Title/Name, designation, material, dimension etc			Article No./Reference	
Designed by ASK		Checked by	Approved by - date	Filename VHalf.dwg	Date 30.04.2022	Scale N/A

USN	4-20mA10V to 0-5V Converters		
	06	Edition 1	Sheet 1/1

Appendix B

Python script for serial communication with Arduino

This paper contains the full Python Script created for serial communication with the Arduino, sketch in Appendix C. The script is created in Jupyter Notebook and contains a brief explanation to each section.

Main_jpyLab

May 18, 2022

1 Python Code for IO controller

1.1 Part of Master's Thesis 2022

Author: Anniken Semb Kvalsund

This file contains the python code used for communication with an Arduino microcontroller configured as an IO-module. The python program communicates with the Arduino through serial bus.

NOTE: Information sent to the Arduino must be bytewise, not as strings, as the Arduino controller reads strings too slow. Capital letters, special symbols and lower case a,b,c and d are reserved for ASCII byte representation of integer values between 0-100.

If access to port denied, close Arduino IDE and try again.

1.2 Main Code

Importing libraries etc:

```
[ ]: #%pylab notebook
     import serial
     import time
     import numpy as np
```

Establishing communication with the serial port. Remember to change the 'COM3' to whichever port is in use. The time.sleep(2) allows for some startup time for the Arduino.

```
[ ]: ser = serial.Serial('COM3', baudrate = 9600, timeout = 1)    # Timeout unit =␣
      ↪seconds
     time.sleep(2)
```

Scaling function An allround useful function for scaling variables

```
[ ]: # Function to convert value to different scales
     def scaleVal(invalue, in_min, in_max, out_min, out_max):
         out = ((invalue - in_min)/(in_max-in_min))*(out_max-out_min)+out_min
         return out

     #print(ScaleVal(0,-10,10,-1000,1000))
```

Defining variables etc. numPoints defines how many data point one wants to collect in one round (based on a for-loop). This will eventually be removed in a final stage of the program. The dataList creates a list for said collected datapoints.

[ ]:
```
numPoints = 20                         # Number of data rows to be collected.␣
 ↪Remove when program is continuously reading.
dataList = np.array([0]*numPoints)    # Create list for data points.
#AIs = [None]*numPoints]         # Analogue input matrix
whileLoop = True
Ch = ['0','1','2','3']

rounds = 0 # For counting round of AI read

AI_1_temp = [None]*numPoints
AI_2_temp = [None]*numPoints
AI_3_temp = [None]*numPoints
```

Sanitising input function Makes the user type in answers until they type in a valid one. Found at: https://stackoverflow.com/questions/23294658/asking-the-user-for-input-until-they-give-a-valid-response

[ ]:
```
def clean_input(prompt, type_=None, min_=None, max_=None, range_=None):
    if min_ is not None and max_ is not None and max_ < min_:
        raise ValueError("min_ must be less than or equal to max_.")
    while True:
        ui = input(prompt)
        if type_ is not None:
            try:
                ui = type_(ui)
            except ValueError:
                print("Input type must be {0}.".format(type_.__name__))
                continue
        if max_ is not None and ui > max_:
            print("Input must be less than or equal to {0}.".format(max_))
        elif min_ is not None and ui < min_:
            print("Input must be greater than or equal to {0}.".format(min_))
        elif range_ is not None and ui not in range_:
            if isinstance(range_, range):
                template = "Input must be between {0.start} and {0.stop}."
                print(template.format(range_))
            else:
                template = "Input must be {0}."
                if len(range_) == 1:
                    print(template.format(*range_))
                else:
                    expected = " or ".join((
                        ", ".join(str(x) for x in range_[:-1]),
                        str(range_[-1])
                    ))
                    print(template.format(expected))
        else:
            return ui
```

1.2.1 Functions

Read analogue inputs The 'AI_read' function reads values from the analogue inputs (A0-A3). The data is transmitted with 10bit resolution.

[ ]:
```
# Function for reading analogue values
def AI_read():
    #ser.write(b'e')                                          # b signifies that␣
 ↪we're writing a byte to the serial bus
    arduinoData = ser.readline().decode().rstrip()          # Reads the arduino␣
 ↪point from the ser port specified above. Readline reads until the end of␣
 ↪line character. ascii decode removes the information around data read (byte␣
 ↪rn)
    list_values = arduinoData.split('x')                    # Splits the multiple␣
 ↪elements by x
#    list_values_int = list(map(int, list_values))         # Converts lists of␣
 ↪strings to list of integers
    return list_values
```

Sort Collected Data in Matrix The function 'sortData' calls the 'AI_read' function, as described above and sorts the arrays into a matrix-form.. Each column in the matrix represents the values collected from one input.

[ ]:
```
#Function to sort data in Matrix
def sortData(AI1, AI2, AI3):
    for i in range(0,numPoints):                        # Limits number of data␣
 ↪transferred. Can be removed later
        ser.write(bytes('e', 'utf-8'))                  # calls for an analogue input␣
 ↪reading
        ser.write(bytes('~', 'utf-8'))
        data = AI_read()                                # Call the Function that reads␣
 ↪the Analogue values
        #print(data)                                     # Prints the recieved data

        # Sorting data from the three different channels to their own array
        AI_1_temp[i] = round(scaleVal(int(data[0]),0,1023,-10,10),3)
        AI_2_temp[i] = round(scaleVal(int(data[1]),0,1023,-10,10),3)
        AI_3_temp[i] = round(scaleVal(int(data[2]),205,1023,4,20),3)

    AI1_return = AI1 + AI_1_temp
    AI2_return = AI2 + AI_2_temp
    AI3_return = AI3 + AI_3_temp

    #print('\nAI 1 = ', AI1_return,'\nAI 2 = ', AI2_return,'\nAI 3 = ',␣
 ↪AI3_return,)
    #print('\nAI 1 = ', AI_1, '\nAI 2 = ', AI_2, '\nAI 3 = ', AI_3)

    return AI1_return,AI2_return,AI3_return
```

Controlling built in Arduino LED The function can switch on and off the built in Arduino

LED. It is not a vital function, but comes in handy during troubleshooting plausible connectin issues.

```
[ ]: # Function for controlling built in Arduino LED
     def LEDonoff(LEDcontr):
         if LEDcontr == 'on':
             ser.write(bytes('h', 'utf-8'))
             ser.write(bytes('~', 'utf-8'))
         elif LEDcontr == 'off':
             ser.write(bytes('i', 'utf-8'))
             ser.write(bytes('~', 'utf-8'))
         arduinoData = ser.readline().decode()
         return arduinoData
```

Limit Function (utils) This function makes sure the number is within a specified range. Default is 0-100

```
[ ]: #Limits input 'num' between minimum and maximum values
     def limit(num, minimum=0, maximum=100):
         limited = max(min(num, maximum), minimum)
         return limited
```

Analog / Digital Out write This function formats the neccessary info and sends it bytewise to the bus. The values are collected form an array separated by x'es and ended by a '~' - The first value in array chooses between digital or analogue channels. 'o' for analogue and 'p' for digital. - The second value specifies the channel number - The third value specifies the value to be sent on said channel. For analogue 0-100, for digital 0 or 1.

```
[ ]: # Function for sending an array of information to serial bus, separated by x'es␣
      ↪and endedn with \n. UTILS
     def byteWriteArray(AD,chNo,chVal):                # Inputs must be characters

         channelAD = bytes(AD, 'utf-8')                # Convert AD (o/p) to byte
         channelNo = bytes(chNo, 'utf-8')              # Convert chNumber to byte
         channelVal = bytes(chVal, 'utf-8')      # Converts ch value to ASCII␣
      ↪character to byte

         fullArray = [channelAD, channelNo, channelVal] # Creates array of␣
      ↪analog,chNumber and value

         for i in range(0, (len(fullArray))):         # Send values for the length og␣
      ↪the array
             ser.write(fullArray[i])                   # Send value in array
             #print(fullArray[i])
             if i <= (len(fullArray)-2):               # Goes to else at second last␣
      ↪element
                 ser.write(bytes('x', 'utf-8'))        # Separate by x between each␣
      ↪element, exept for after the last
             else:
```

```
            ser.write(bytes('~', 'utf-8'))          # End with new line char ␣
↪
```

Print Results – NOT needed, useful when troubleshooting. – This function simply prints the feedback and its type sent from the arduino.

[ ]:
```
# This function is for testing purposes only
def printResults():
    feedback = ser.readline().decode().rstrip()  # Reads the arduino point from␣
↪the ser port specified above. Readline reads until the end of line character.
↪ ascii decode removes the information around data read (byte rn)
    print('Recieved arduino Data: ',feedback)
    print('Recieved data type: ',type(feedback),'\n\n')
```

Arduino max AO cal This functio is made for ard. AO calibration. The PWM AO should ideally put out about 5V, but the real voltage is usually a bit higher. The function is based on arduinos maximum out value being 255, and adjusts this down accordingly to create exactly 5V out. To avoid too large numbers on the serial bus (should ideally be between 0 and 100, sent as byte) the function finds the difference between the desired output(10V) and real output, multiplies it by 100 to avoid decimals and sends this value to the arduino. The measured voltage should be between 10 and 11. If outside this range, other measures must be take to correct it anyways. The function is based on:

$$ArduinoAO_{max,new} = \frac{V_{max,desired}}{\frac{V_{max,real}}{Arduino_{max}}} = \frac{10V}{\frac{10.xV}{255}}$$

[ ]:
```
# This funtion is used to adjust the maximum output value for arduino PWM␣
↪outputs

def arduinocal(Vmeas_max,Vmeas_min,Vdesired):
    i = 0
    Vclosetomin = False   # Used to reset and try again is voltage is too low
    Vreset = False # Used to reset max value
    sendValue = 0  # Difference value to be sent to Arduino

    AOchannels = ['AO_0','AO_1','AO_2','AO_3']    # Array of analogue output␣
↪channels
    print(' - Press enter to go to next channel. \n - Type reset to reset␣
↪channel value.\n')

    for i in range(0, (len(AOchannels))):
        Vreset = False # Resets the Vreset variable for every channel.

        while True: # Only accepts numbers as input, loops to avoid errors.
            Vmeas_str = input('Measured output voltage channel {}: '.
↪format(AOchannels[i]))

            if not Vmeas_str:   # If user presses "enter", then break loop and␣
↪go to next channel
```

```
            break
        else:
            if Vmeas_str == 'reset' or Vmeas_str == 'Reset':  # Reset␣
↪maximum value
                Vreset = True
            else:
                try:
                    Vmeas_float = float(Vmeas_str)    # Ensures the input␣
↪is a number
                except ValueError:
                    print('Not a valid number. Use . as decimal symbol.')
                    continue

        if (not Vreset) and (Vmeas_float > Vmeas_max or Vmeas_float <␣
↪(Vmeas_min-1)):   # Checks if the number is (not) within the right range
            print('Find other source of adjustment. Voltage deviance too␣
↪high.')
            break

        else:
            if (not Vreset) and (Vmeas_float > (Vmeas_min-1) and␣
↪Vmeas_float < Vmeas_min): # If measured value is between 9-10, reset
                Vreset = True
                Vclosetomin = True
                print('Channel ', AOchannels[i],' is too low. Max value␣
↪reset to default.\nPlease try again.')

            if Vreset == True:
                sendValue = 'r'
            else:
                dev_val = int(round((Vmeas_float - Vdesired)*100))  #␣
↪Calculates deviation between desired and actual value
                #print("Deviation 0-100 = ",dev_val)
                sendValue = str(chr(dev_val))

            byteWriteArray('q', str(i), sendValue) # Change the max value
            byteWriteArray('o', str(i), 'd') # Set said channel to max

            if Vclosetomin == True:  # Resets Vclosetomin variable and runs␣
↪the loop once more if value between 9 and 10.
                Vclosetomin = False
                continue
            else:
                break
```

Main This is the main function of the program. The user is asked which function they would like to run, and the if-statements calls the respective function. In a final version, the read functions will run continuously, and the write will run when changed.

[ ]:
```
#%% Main program

def application():

    whileLoop = True

    AI_0 = []
    AI_1 = []
    AI_2 = []

    AI0tmp = []  # Placeholders for AI lists
    AI1tmp = []
    AI2tmp = []

    modes = ['read','write','cal','on','off','q','Q'] # Modes/programs


    #Whileloop:
    while whileLoop == True:

        print('\nGet Data? \n - ',modes[0],' = analogue input values \n -␣
↪',modes[1],' = Write to analogue or digital out \n - ',modes[2],' =␣
↪Calibrate arduino analogue outputs. \n - ',modes[3],' = LED on \n -␣
↪',modes[4],' = LED off \n - ',modes[5],' = Close Port \n')

        # Asking which function should be rund
        chooseMode = clean_input('Please choose mode: ', range_=modes)


        # Get datapoints from analogue inputs A0, A1, A2 and A3
        if chooseMode == modes[0]:   # If user press AI, get datapoints
            AI0tmp = AI_0
            AI1tmp = AI_1
            AI2tmp = AI_2
            AI_0, AI_1, AI_2 = sortData(AI0tmp, AI1tmp, AI2tmp)

            print('\nAI 0: ',AI_0,'\nAI 1: ',AI_1,'\nAI 2: ',AI_2,'\n')

        # Analogue/digital write
        if chooseMode == modes[1]:

            # Ask if write to analogue or digital out
            ADInput = clean_input('Select A for analogue, or D for digital:␣
↪',range_=('A','a','D','d')) # Choose between analogue and digital channels
            chNumberInput = clean_input('Select channel number: ',range_=Ch) #␣
↪Write desired channel number.
```

```
            # Analogue write out
            if (ADInput == 'A' or ADInput == 'a') and chNumberInput in Ch:

                numberinput = clean_input('Insert value 0-100: ', type_=int,␣
↪min_=0, max_=100) # Ask for value between 0-100 to send to ard

                byteWriteArray('o', chNumberInput, str(chr(numberinput))) #␣
↪chr=int to ASCII, then converted to string


            # Digital write out
            elif (ADInput == 'D' or ADInput == 'd') and chNumberInput in Ch:

                onoffin = clean_input('True or False?',␣
↪range_=('True','true','on','1','False','false','off','0')) # Ask if DO ␣
↪should be high or low

                if onoffin == 'True' or onoffin == 'true'  or onoffin =='on' or␣
↪onoffin == '1':# Checks if true
                    onoff = 60

                elif onoffin == 'False' or onoffin == 'false'  or onoffin␣
↪=='off' or onoffin == '0':# Checks if false
                    onoff = 0

                    byteWriteArray('p', chNumberInput, str(chr(onoff)))

                else:
                    print('Something is wrong\n')
            else:
                print('Invalid input. Channel must be A or D, and number␣
↪between 0-3\n\n')


        if chooseMode == modes[2]:     # Calibration:

            enterCalMode = clean_input('Entering calibration mode. Please make␣
↪sure equipment is disconnected before continuing.\nDo you want to proceed? Y/
↪N: ',range_=('Y','y','N','n'))


            if enterCalMode == 'Y' or enterCalMode == 'y':
                for k in range(0, 4):
                    byteWriteArray('o', str(k), 'd') # Setting all the AO to␣
↪max
```

```
                print('\nCalibration mode. Please measure the output analogue␣
↪output one by one.')
                arduinocal(11,10,10)

        # Switch builtin LED on/off
        if chooseMode == modes[3] or [4]:
            LEDonoff(chooseMode)

        if chooseMode == modes[5] or chooseMode == modes[6]:
            print('Serial port closed.')
            ser.close()
            whileLoop = False
```

If name is main This runs the main function

[ ]:
```
# %% Running
if __name__ == '__main__':
    application()
```

Appendix C

Arduino sketch for serial communication with Python

This paper contains the full Arduino sketch created for serial communication with the Python script in Appendix B. The script is created in Arduino IDE and contains brief explanations as comments.

Main .ino

May 18, 2022

[ ]:
```
/* Sketch for ARDUINO communication with Python.
 *
 *  Part of masters thesis spring 2022.
 *  Anniken Semb Kvalsund
 *  Electrical Power Engineering.
 *
 */

// Constants
const int AI[3] = {A0, A1, A2}; // Creates an array of all the analog inputs
byte noAI = (sizeof(AI)/sizeof(AI[0]));   // Finds the number of elements in AI␣
 ↪array. To use in for loops etc

// Variables
int data1 = 0;          // Input information. Initialise to zero.
int i = 0;

bool readAI = false;
bool writeADO = false;
bool calibrateAO = false;
bool LEDcontr = false;

int AIs[10];            // For "storing" collected analogue input values
char AIstring[16];      // For storing AI values converted to string

// Write data constants and variables
const byte numChars = 16;
char receivedChars[numChars];   // an array to store the received data
byte ndx = 0;
boolean newData = false;
char analogDigital;
int channel;   // Converts the number recieved in channel number to an int
int chValConv; // Sends the char chVal to string toInt function, returned␣
 ↪scaled and converted to int.
int outVal; // Output value to analogue PWM outputs.
bool LEDon = false; // LED controlling variable.
```

```
// Maxvalue to PWM outputs. Adjustable to ensure 5V, not more, is output at max.
int max_out_AO0 = 255;
int max_out_AO1 = 255;
int max_out_AO2 = 255;
int max_out_AO3 = 255;

// Declare outputs
const int DO0 = 2;
const int DO1 = 4;
const int DO2 = 7;
const int DO3 = 8;

const int AO0 = 3;      // 0-10V
const int AO1 = 5;      // 0-10V
const int AO2 = 6;      // +-10V
const int AO3 = 9;      // +-10V


void setup() {
  // Setting up an initialise the serial communication
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(DO0, OUTPUT);
  pinMode(DO1, OUTPUT);
  pinMode(DO2, OUTPUT);
  pinMode(DO3, OUTPUT);
  pinMode(AO0, OUTPUT);
  pinMode(AO1, OUTPUT);
  pinMode(AO2, OUTPUT);
  pinMode(AO3, OUTPUT);

}



void loop() {
if(Serial.available()>0){           // Return the number of bytes available on␣
 ↪serial. if <0, = no info on serial.

  recvWithEndMarker();

  if(readAI == true){
    analoginputs();
    readAI = false;
  }

  if(writeADO == true){
    analogueOut();
    writeADO = false;
  }

  if(calibrateAO == true){
    calibrateAO_func();
    calibrateAO = false;
  }

  if (LEDcontr == true){
    LEDonoff();
    LEDcontr = false;
  }

  else{/* Do nothing*/}
  } // if serial available
} // Void loop

//
  ↪-------------------------------------------------------------------------------------------
// Read analogue inputs

void analoginputs(){
  for ( i=0; i<=noAI; i++){
     AIs[i] = analogRead(AI[i]);
   } // for AI

   // Converting the analog input values to a single string with x's separating␣
  ↪each value.
   sprintf(AIstring, "%dx%dx%d" , AIs[0],AIs[1],AIs[2]);
   Serial.println(AIstring);       // Writes data from AI to serial bus
} // void analoginputs


//
  ↪-------------------------------------------------------------------------------------------
// Reads from Serial port until endChar '~' is received.

void recvWithEndMarker() {
  char endMarker = '~';
  char rc;
  boolean done = false;

  while (Serial.available() && !done) {
    rc = Serial.read();
    if (rc == endMarker) {
      done = true;
      newData  = true;
    }
    else {
      receivedChars[ndx++] = rc;
      if (ndx >= numChars)
        done = true;
    }
  }

  if (newData) {
    if (!parseInput()) { // if we call parseData it fails there is no need to␣
 ↪process that data further.
      //Serial.println("String couldn't be parsed");
      newData = false;
    }
    ndx = 0;
  }
}


//
 ↪-------------------------------------------------------------------------------------------
// Splits the received string into its separate characters and stores them in␣
 ↪globals

boolean parseInput() {
  int secondX = -1;
  String tempString;
  String chValConvtemp;
  char Buf[2];      // For converting string to char


  for (int i = 0; i < ndx; i++)
    //Serial.print(receivedChars[i]);
    //Serial.println();

  if (receivedChars[1] != 'x'){   // If no further values are read, ie not␣
 ↪split by x'es
     if (receivedChars[0] == 'e'){ readAI = true;} // The readAI value is set␣
 ↪to true, causing readAI function to run
     if (receivedChars[0] == 'h'){ LEDcontr = true; LEDon = true;}
     if (receivedChars[0] == 'i'){ LEDcontr = true; LEDon = false;}
    return false;}
  else {                          // If more than one value is transmitted

    if (receivedChars[0] == 'o' or receivedChars[0] == 'p'){ writeADO = true;␣
 ↪analogDigital = receivedChars[0];} // Prepare to write values
```

```
    if (receivedChars[0] == 'q'){ calibrateA0 = true;}   // Prepare to␣
 ↪calibrate (using the same message system as writeAD0

    for (int i = 3; i < ndx && secondX == -1; i++)
      if (receivedChars[i] == 'x')
        secondX = i;

    if (secondX == -1)
      return false;

    tempString = receivedChars;
    channel = tempString.substring(2, secondX).toInt();

    chValConvtemp = tempString.substring(secondX + 1, ndx);
    chValConvtemp.toCharArray(Buf, 2);
    chValConv = int(Buf[0]);

// Print for troubleshooting
    Serial.print("analogDigital -> ");
    Serial.println(analogDigital);
    Serial.print("channel -> ");
    Serial.println(channel);
    Serial.print("chValConv -> ");
    Serial.println(chValConv);
  }
  return true;
}

//
 ↪--------------------------------------------------------------------------------------------
// Write analogue and / or digital values

void analogueOut() { // Writes to analogue outputs

  if (newData == true) {
    // Result in an array with two elements. One A1/D2 etc and one Value/onoff

    if (analogDigital == 'o') {
      //Serial.println("Analogue");

      switch (channel) {
        case 0:
          //Serial.println("A00");
          outVal = constrain((map(chValConv, 0, 100, 0, max_out_A00)), 0,␣
 ↪max_out_A00);
          analogWrite(A00, outVal);
          break;
        case 1:
          //Serial.println("AO1");
          outVal = constrain((map(chValConv, 0, 100, 0, max_out_AO1)), 0,␣
↪max_out_AO1);
          analogWrite(AO1, outVal);
          break;
        case 2:
          //Serial.println("AO2");
          outVal = constrain((map(chValConv, 0, 100, 0, max_out_AO2)), 0,␣
↪max_out_AO2);
          analogWrite(AO2, outVal);
          break;
        case 3:
          //Serial.println("AO3");
          outVal = constrain((map(chValConv, 0, 100, 0, max_out_AO3)), 0,␣
↪max_out_AO3);
          analogWrite(AO3, outVal);
          break;
        default:
          //Serial.println("NaN");
          break;
      } // Switch case channel

    } // if recievedArray[0]=o

    else if (analogDigital == 'p') {
      //Serial.println("Digital");
      bool dContr = false;

      // Creating a "buffer"
      if (chValConv < 50){dContr = LOW;}
      else if (chValConv >= 50){dContr = HIGH;}
      else {dContr = LOW;}

      switch (channel) {
        case 0:
          //Serial.println("DO0");
          digitalWrite(DO0, dContr);
          break;
        case 1:
          //Serial.println("DO1");
          digitalWrite(DO1, chValConv);
          break;
        case 2:
          //Serial.println("DO2");
          digitalWrite(DO2, chValConv);
          break;
        case 3:
          //Serial.println("DO3");
          digitalWrite(DO3, chValConv);
          break;
        default:
          //Serial.println("NaN");
          break;
      } // Switch case channel

    } // if recievedArray[0]=p

    else {
      //Serial.println("Channel not valid");
    } // if recievedArray[0]=p

    newData = false;
  } // if newData True
} // void analogueOut


//
 ↪-----------------------------------------------------------------------------------------

// Function for converting the recieved value 0-100 to calibration values, or␣
 ↪reset cal. vlaues.
int calEq(int range, int measVal, int maxOut){
  int maxOut_tmp; // For storing the value temporary

  if (measVal == 114){  // If measVal is 114, aka reset, the max \Out is set to␣
 ↪default 255
    maxOut_tmp = 255;
    } // if measVal 114('r')
  else { // If measVal is something else, the recieved value(measVal), scales␣
 ↪it to measured voltage(10-11V), corrects the offset and scales the new value␣
 ↪to out max
    maxOut_tmp = int(round(range/((range+(float(measVal)/100))/maxOut)));
  } // if measVal is not 114('r')

  maxOut = maxOut_tmp;

  return maxOut;
  }

//
 ↪-----------------------------------------------------------------------------------------

void calibrateAO_func() { // Calibrate analogue PWM outs
```

```
  int range = 0; // Range of values. Set to 10 or 20, depending on output␣
↪channel
  if (channel == 0 or channel == 1){range = 20;} // For AO_0 and AO_1 with -10␣
↪to +10V range
  else {range = 10;} // For AO_2 and AO_3 with 0-10V range.

 // Converted new maximum values for all four channels
 int new_max_AO0;
 int new_max_AO1;
 int new_max_AO2;
 int new_max_AO3;

  if (newData == true) {
    switch (channel) {
      case 0:
        new_max_AO0 = calEq(range, chValConv, max_out_AO0);
        max_out_AO0 = new_max_AO0;
        break;
      case 1:
        new_max_AO1 = calEq(range, chValConv, max_out_AO1);
        max_out_AO1 = new_max_AO1;
        break;
      case 2:
        new_max_AO2 = calEq(range, chValConv, max_out_AO2);
        max_out_AO2 = new_max_AO2;
        break;
      case 3:
        new_max_AO3 = calEq(range, chValConv, max_out_AO3);
        max_out_AO3 = new_max_AO3;
        break;
      default:
        //Serial.println("NaN");
        break;
    }  // Switch case channel

    newData = false;
  } // if newData True
}


void LEDonoff(){
  if (LEDon == true) {digitalWrite(LED_BUILTIN, HIGH);}   // switch the LED on}
  //else {digitalWrite(LED_BUILTIN, LOW);}   // turn the LED off}
} // Woid LEDonoff
```

Appendix D

Current, voltage and DAC/ADC converter results

This paper includes the measured results from the four analogue converters and DAC/ADC:

- *Voltage Doubler*: $0-5V$ to $0-10V$,
- *Voltage Quadrupler*: $0-5V$ to $\pm10V$
- *Voltage Quarter*: $\pm10V$ to $0-5V$
- *Current to Voltage converter*: $4-20mA$ to $0-5V$
- *Analogue to Digital Converter*: $0-5V$ to $10-bit$
- *Digital to Analogue Converter*: $8-bit$ to $0-5V_{\text{PWM}}$

The paper shoes the full, raw data collected from the converter tests, after calibration. The input voltage or current was provided by a generic DC supply connected to their input terminals, where they would later be connected to the Arduino. The output voltage was measured using a handheld multimeter connected to the converters' output terminals.

Table D.1: $0-5V$ to $0-10V$ converter results

Channel 1		Channel 2	
In [V]	**Out**[V]	**In** [V]	**Out**[V]
0.00	0.00	0.00	0.00
0.25	0.51	0.25	0.50
0.50	1.01	0.50	1.00
0.75	1.50	0.75	1.50
1.00	1.98	1.00	1.99
1.25	2.46	1.25	2.49
1.50	2.94	1.50	2.99
1.75	3.43	1.75	3.49
2.00	3.93	2.00	3.99
2.25	4.43	2.25	4.49
2.50	4.93	2.50	4.98
2.75	5.44	2.75	5.48
3.00	5.94	3.00	5.98
3.25	6.45	3.25	6.48
3.50	6.95	3.50	6.98
3.75	7.46	3.75	7.49
4.00	7.97	4.00	7.99
4.25	8.47	4.25	8.49
4.50	8.97	4.50	8.99
4.75	9.48	4.75	9.49
5.00	9.99	5.00	9.99

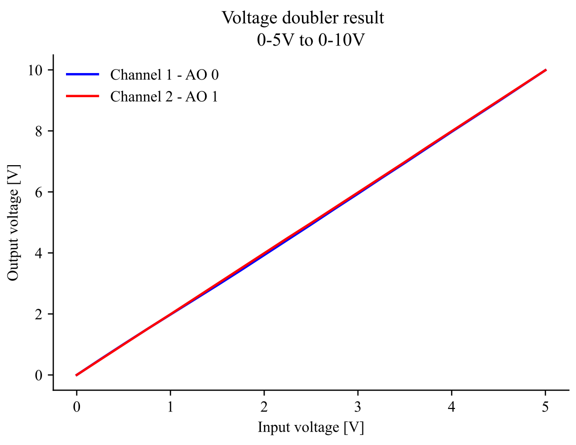


Figure D.1: Analogue voltage doubler output

Table D.2: $0-5V$ to $\pm 10V$ converter results

Channel 1		Channel 2	
In [V]	**Out**[V]	**In** [V]	**Out**[V]
0.00	-10.00	0.00	-10.00
0.25	-9.00	0.25	-9.00
0.50	-7.99	0.50	-8.00
0.75	-6.98	0.75	-7.00
1.00	-5.97	1.00	-6.00
1.25	-4.97	1.25	-5.00
1.50	-3.96	1.50	-4.00
1.75	-2.96	1.75	-3.00
2.00	-1.96	2.00	-2.00
2.25	-0.95	2.25	-1.00
2.50	0.02	2.50	0.00
2.75	1.03	2.75	1.00
3.00	2.03	3.00	2.00
3.25	3.03	3.25	3.00
3.50	4.03	3.50	3.99
3.75	5.02	3.75	4.99
4.00	6.03	4.00	5.99
4.25	7.03	4.25	7.00
4.50	8.02	4.50	7.99
4.75	9.01	4.75	8.99
5.00	10.00	5.00	10.00

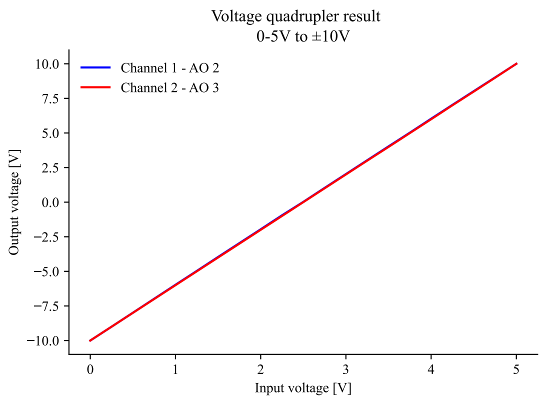


Figure D.2: Analogue voltage quadrupler output

Table D.3: $\pm 10V$ to $0-5V$ converter results

Channel 1		Channel 2	
In [V]	**Out**[V]	**In** [V]	**Out**[V]
-10.00	0.01	-10.00	0.03
-9.00	0.26	-9.00	0.27
-8.00	0.51	-8.00	0.40
-7.00	0.76	-7.00	0.76
-6.00	1.01	-6.00	1.01
-5.00	1.25	-5.00	1.26
-4.00	1.50	-4.00	1.50
-3.00	1.75	-3.00	1.75
-2.00	2.00	-2.00	2.00
-1.00	2.25	-1.00	2.24
0.00	2.50	0.00	2.49
1.00	2.74	1.00	2.74
2.00	2.99	2.00	2.98
3.00	3.24	3.00	3.23
4.00	3.48	4.00	3.46
5.00	3.73	5.00	3.71
6.00	3.97	6.00	3.96
7.00	4.22	7.00	4.20
8.00	4.47	8.00	4.45
9.00	4.72	9.00	4.70
10.00	4.97	10.00	4.95

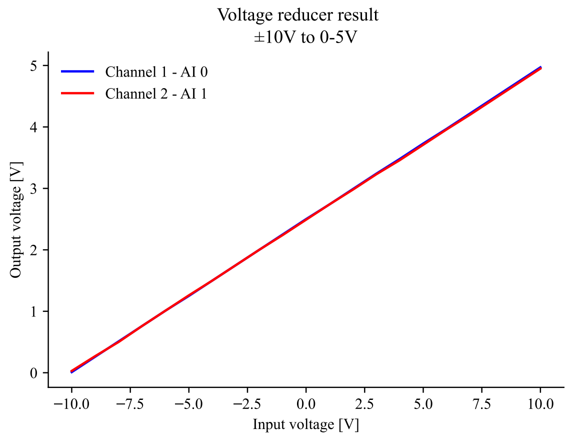


Figure D.3: Analogue voltage quarter output

Table D.4: $4-20mA$ to $1-5V$ converter results

Channel 1	
In [V]	**Out**[V]
4.00	1.01
5.00	1.26
6.00	1.50
7.00	1.74
8.00	2.01
9.00	2.24
10.00	2.49
11.00	2.74
12.00	2.99
13.00	3.24
14.00	3.49
15.00	3.76
16.00	4.01
17.00	4.25
18.00	4.51
19.00	4.76
20.00	5.01

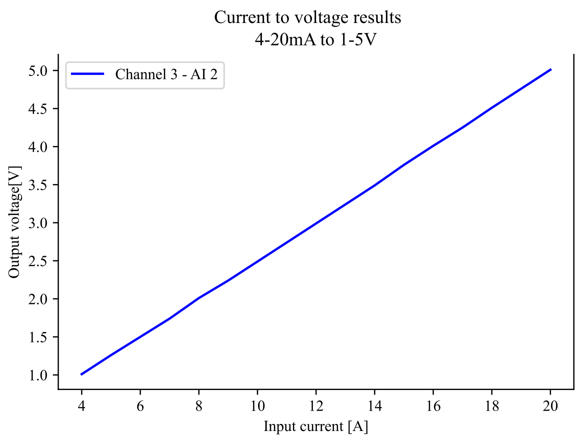


Figure D.4: Analogue current to voltage converter results

Table D.5: $0-5V$ to 10bit Arduino converter results

Channel 1		Channel 2	
In [V]	**Received [10bit]**	**In [V]**	**Received [10bit]**
0	0	0	0
0.01	1	0.03	4
0.26	51	0.27	53
0.51	100	0.40	102
0.76	150	0.76	152
1.01	199	1.01	201
1.25	250	1.26	250
1.50	299	1.50	300
1.75	349	1.75	349
2.00	399	2.00	399
2.25	449	2.24	449
2.50	498	2.49	499
2.74	549	2.74	548
2.99	599	2.98	597
3.24	649	3.23	647
3.48	699	3.46	696
3.73	748	3.71	746
3.97	798	3.96	795
4.22	848	4.20	844
4.47	897	4.45	893
4.72	947	4.70	943
4.97	998	4.95	992
5.01	1004	5.00	1002

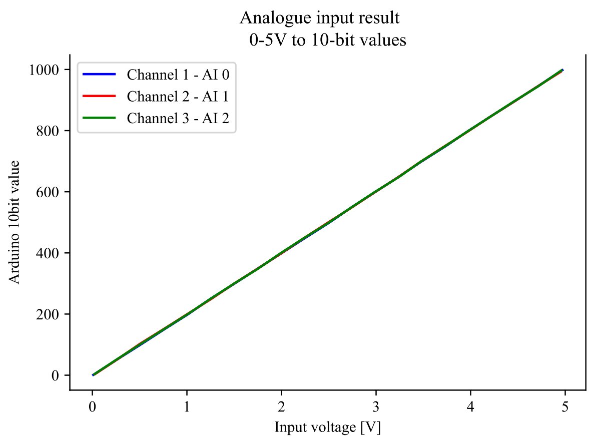


Figure D.5: $0-5V$ to Arduino 10bit converter

Table D.6: 8bit Arduino to $0-5V$ converter results

	Analogue PWM output [V]			
Command (8bit)	AO_0	AO_1	AO_2	AO_3
0	0.00	0.00	0.00	0.00
10	0.20	0.22	0.22	0.20
20	0.40	0.42	0.42	0.42
30	0.61	0.62	0.62	0.60
40	0.81	0.82	0.82	0.80
50	1.01	1.02	1.02	1.01
60	1.21	1.22	1.22	1.21
70	1.41	1.42	1.42	1.41
80	1.60	1.62	1.62	1.61
90	1.81	1.82	1.82	1.81
100	2.01	2.02	2.02	2.01
110	2.21	2.22	2.22	2.21
120	2.41	2.42	2.42	2.41
130	2.61	2.62	2.62	2.61
140	2.81	2.82	2.82	2.81
150	3.01	3.02	3.02	3.01
160	3.20	3.20	3.20	3.21
170	3.40	3.40	3.40	3.41
180	3.60	3.60	3.60	3.61
190	3.80	3.80	3.80	3.81
200	4.00	4.00	4.00	4.01
210	4.20	4.20	4.20	4.21
220	4.40	4.40	4.40	4.41
230	4.60	4.60	4.60	4.61
240	4.80	4.80	4.80	4.81
250	5.00	5.00	5.00	5.01
255	5.11	5.11	5.11	5.11

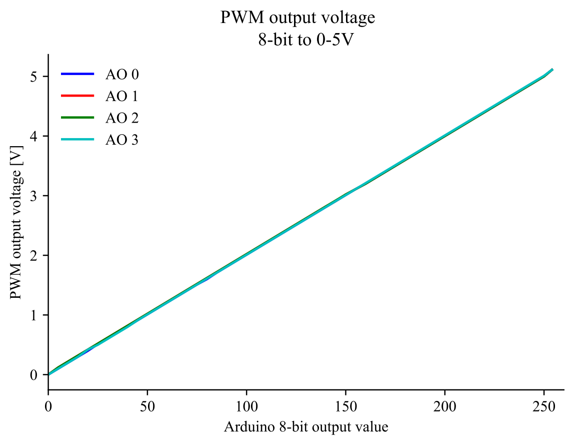


Figure D.6: Arduino 8bit to $0-5V$ converter

Appendix E

Lenze 9400 Servo Drive complete function block diagram

This paper contains the complete function block diagram implemented in the Lenze Highline 9400 Servo Drive when integrated to the Lucas Nülle Test Stand interface. The diagram is exported form the drive using the software Lenze Engineer. [28]

08.05.2022 Stellantrieb - Drehzahl