CODEN:LUTEDX/(TEIE-5513)/1-90/(2024)

Industrial Electrical Engineering and Automation

DC Motor Control without Encoder – A Ripple Based Approach

Axel Björnberg
Jonathan Vesterlund

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

DC Motor Control Without Encoder - A Ripple Based Approach

Axel Björnberg & Jonathan Vesterlund

LUND
UNIVERSITY

Department of Industrial Electrical Engineering & Automation

MSc Thesis TEIE-5513

Department of Industrial Electrical Engineering & Automation
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Media-Tryck.
Lund 2024

Abstract

The goal of this master thesis was to identify an alternative to shaft encoders as means of achieving accurate motor velocity and position control for sliding door systems. While the company had previously explored methods like using back-EMF, this thesis investigates counting voltage ripples during the commutation process in a brushed DC motor. During commutation, one or two pairs of commutator segments contact the brushes simultaneously, periodically changing the motor's impedance. With a constant current, this creates a ripple observable on the voltage.

Through a series of experiments, it was possible to produce a clear ripple signal originating from the commutator commutation. This was done even though high frequency switching of the internal H-bridge used for motor control was utilized. By employing a series of filters and amplifiers, coupled with Arduino microcontrollers, noise could be removed and the wanted signal amplified. The signal was then used to produce a square wave representing the commutation ripple. This square wave could then be read by an Arduino using an interrupt pin.

For the software solutions, a moving average, a finite state machine (FSM), and a solution to remove multiple readings of a single ripples were used. These software solutions was used in combination with the circuitry to further increase the accuracy of the detection and counting of voltage ripple.

The resulting solution provided mostly correct velocity estimations as compared to the onboard encoder. The difference being the lower resolution and some difficulties in recording all ripples. The performance was adequate at low to medium velocities, but started to deviate at higher velocities as a result of, amongst other factors, inadequate sampling rates. Another problem that was observed was the loss of observable ripples during deceleration. This was theorized to be caused by the way the signal was sampled from the H-bridge, whereas a differential reading across the bridge might render better results.

This report shows that voltage ripples can track velocity and position. While the current solution is insufficient for ripple-based motor control, future research based on the produced findings might enable its implementation in the future.

Acknowledgements

Throughout our master thesis journey there have been many who supported us in numerous ways. This page is dedicated to extending our gratitude to those people.

First of all, we would like to thank our supervisor at Lunds Tekniska Högskola, Samuel Estenlund, for his support in our academic work. We are especially grateful for his guidance during the first crucial weeks, where he helped us find our footing. We would also like to take a moment to thank our examiner, Avo Reinap, for dedicating a piece of his time to ensure that our project met the appropriate academic standards.

We would also like to acknowledge the immense assistance we received from employees and supervisors at ASSA ABLOY Entrance System in Landskrona. This thesis was made possible thanks to their support. Special thanks goes out to our project supervisors, Roger Dreyer and Jerker Örjmark, for introducing us to the problem at hand and providing us with crucial information and help throughout the duration of our thesis. We are also particularly grateful to Fredrick Brodje and Felix Bille for their support, expertise and suggestions during our work which helped propel our work forward. Additionally, we would like to thank Louise Bannersten and Elin Elfström for their work with the student initiatives at ASSA ABLOY and for helping make our time at the company unforgettable.

Without the contributions and support of the individuals mentioned above, as well as others, this thesis would not have been possible. Thank you!

List of Abbreviations

ADC: Analog-to-digital converter

DAC: Digital-to-analog converter

SAMD: A family of micro-controllers developed by Microchip Technology

ISR: Interrupt service routine

MOSFET: Metal–oxide–semiconductor field-effect transistor

DMA: Direct memory access

FSM: Finate state machine

PWM: Pulse-width modulation

FFT: Fast fourier transform

Contents

1

Introduction

In this chapter the subject of the thesis is described. The background and problem description are also presented along with a clearly formulated objective of the thesis.

1.1 Background

When designing sliding door entrance systems for pedestrian use, the safety of the user is of great importance. The system has to be able to respond quickly to changes in its environment and be able to precisely control the acceleration, velocity and position of the door leaves. The need for precision control necessitates the use of sensors and techniques that are able to accurately relay safety-critical information. Such systems ensures smooth operation and predictable behavior. The cost of these systems, however, are often not negligible, and it is in the interest of the producers to try to minimize this cost without adversely affecting the safety of the product.

This thesis is being conducted in close cooperation with ASSA ABLOY Entrance Systems in Landskrona. The pedestrian sliding doors designed by ASSA ABLOY utilizes industry-standard brushed DC motors to drive the door leaves via a belt drive. Mounted on the motor shaft are encoders with significant resolution. These encoders are not cheap, and ASSA ABLOY is therefore interested in exploring alternative means of position tracking that would allow them to lower the overall cost of the system.

1.2 Problem description

- Is it possible to implement an adequate control system for a sliding door entrance system without using an encoder?
- How are accurate measurements of the doors position and velocity to be made?

- How are these above points to be implemented by using the existing brushed DC motor?

1.3 Objective

The goal of this thesis is to explore an alternative method to shaft encoders for position control of sliding door systems. By researching other techniques for position control of brushed DC motors, the aim is to try to implement one such technique on an existing sliding door system at ASSA ABLOY Entrance Systems in Landskrona. During the time frame of the project, it is not necessary for the alternative positioning solution to be used to control the door system. Instead, the objective is to demonstrate that the solution can be integrated into the control system with relative ease. The solution should also be cost effective, so as to be able to compete with existing encoder solutions.

1.4 Division of labour

The project's division of labour can be seen as equal overall, with both participants contributing in developing both software and hardware. Additionally, both participants equally contributed to the research, documentation, and to the report.

2

Theory

The following chapter aims to give a comprehensive overview of the theory, hardware and software that form the basis of this project. The theory presented here will later be used throughout the experimental work.

2.1 The brushed DC motor

The motor of interest in this thesis is a permanent-magnet brushed DC motor. Such brushed DC motors have become popular in applications where efficient space utilization is required and where low torques are acceptable [1]. The brushed DC motors utilized in ASSA ABLOY sliding doors are coupled with gearboxes that enable greater output torque at low velocities by allowing the motor to operate at higher velocity.

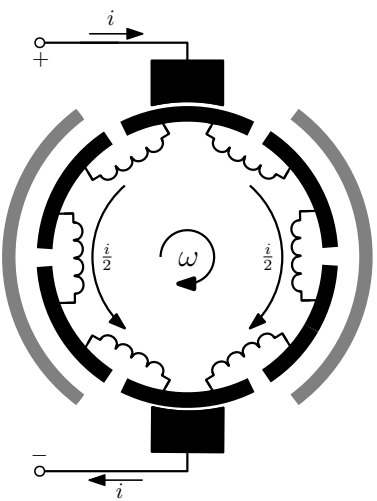

Figure 2.1 A simplified overview of the general permanent-magnet brushed DC motor.

The general brushed DC motor, illustrated schematically in Figure 2.1, utilizes a fixed magnetic field from two permanent magnets on either side of the stator. This magnetic field interacts with the rotor which has copper coils carrying DC current. The ends of the coils are connected to commutator segments on the commutator. As the motor rotates, the brushes periodically contacts new pairs of commutator segments, creating an alternating magnetic field through the coils. This produces a continuous torque towards the permanent magnets, driving the motor [2].

As discussed in the literature related to the course in power electronics at LTH [3], the equation of the armature voltage and the associated motor torque can be given as:

$$u_a = R_a \cdot i_a + L_a \cdot \frac{di_a}{dt} + \omega_r \cdot \Psi_m \tag{2.1}$$

$$T = \Psi_a \cdot i_a \tag{2.2}$$

Where R_a is the armature resistance, i_a is the armature current, L_a is the armature inductance, ω_r is the angular velocity of the rotor given in rad/s and Ψ_m is the magnetic flux of the permanent magnets.

2.2 Encoders

When it comes to position and velocity tracking, shaft encoders are a common solution in many applications. By converting angular rotation to a set number of digital pulses they provide an easy means of position tracking. The encoders commonly used in ASSA ABLOY sliding doors are so called incremental encoders. This means that they respond to change, but do not keep track of the absolute value of pulses. The incremental encoders used are also bi-directional in the sense that the direction of rotation can be deduced from the digital signals.

In general this is achieved by utilizing a mechanical or electrical transducer that continuously reads a pattern on a moving object, in this case a disk. Most incremental rotary encoders use optical light and a transparent disk with black sections to detect these movements. The light-emitting diode beneath the disk shines through, and a photo-detector captures the light, generating electrical pulses as the disk rotates. The system counts these pulses and tracks the shaft position by dividing the number of counted pulses by the known quantity of pulses to calculate the position and the total number of revolutions. For applications requiring detection of direction, a more sophisticated solution is used with a two-channel setup. The two channels are phased 90 degrees apart from each other, allowing the system to

determine the rotational direction by analyzing their phase relationship [4].

The main reason behind the choice of utilizing this kind of encoder has to do with the fact that it is an extremely well tested and robust technique. This is especially important in safety critical applications such as sliding doors meant for pedestrian use. A benefit of encoders is the high precision that can be achieved based on the given resolution. For example, it is not uncommon today to see encoders that yield 500 pulses of data per rotation. This accuracy and simplicity helps motivate the general usage of encoders in motion control systems. To emphasise, an incremental encoder with 500 pulses/rotation could, given a door opening width of only 2 meters, produce over 50 thousand pulses.

2.2.1 Disadvantages

The use of encoders is not without disadvantages. One of the problems is the general size of the sensor. It often makes up a significant percentage of the total motor length if mounted on the motor axle. In applications constrained by limited space, this becomes especially critical. Another problem of using encoders is the general cost of the device. This is closely related to the resolution of the encoder, with more high resolution devices costing more. In the context of sliding doors, the cost of the encoder is the driving factor of limiting encoder use or attempts at lowering the resolution of the device.

2.3 Previous work

Previous work has been conducted at ASSA ABLOY in exploring alternative methods of position tracking. In particular methods of utilizing back-EMF has been made with decent results regarding velocity tracking. The problem with utilizing this approach for position tracking as well is that it is prone to drift, and that achieving sufficient position estimation requires an accurate mathematical motor model.

Implementing such a motor model is not easy, however. This is because of the fact that such a model would be dependant on motor constants such as resistance, inductance and the back-EMF constant. These parameters are not constant but instead vary during operation. This is in part due to heat generated in the motor. As such, even though the motor velocity can still be estimated dynamically, it will lead to a complicated non-linear model [5].

Even if a decent velocity estimation could be achieved with a sufficient non-linear model, drift will always be an issue when constructing a position estimation from a given velocity curve. This has to do with the fact that the position is achieved by integrating the velocity v (Equation 2.3) over time, and any deviation in the estimated velocity will lead to an increasing error term. This is usually fine for

velocity tracking as no consideration has to be made towards the growth of this term, but when it comes to position tracking, not knowing the value of this error term will result in drift in the resulting position value. This is shown in the simplified formula below, where the position $x(t)$ is the sum of the initial position x_0, the integral of the velocity estimation v_{est} over time and the integral of the error over time.

$$x(t) = x_0 + \int_0^t v_{est}(t)\, dt + \int_0^t (v_{true}(t) - v_{est}(t))\, dt \tag{2.3}$$

2.4 Alternative control techniques

In general, sensorless velocity measurement methods for DC motors fall into two divisions. Either they are model based, or they are ripple based. The model based techniques are associated with a high degree of uncertainty given real world conditions, whilst ripple based techniques measures rotor velocity based upon a periodic phenomenon [6]. In this section, a model based approach based upon measurement of the back-EMF as well as ripple based techniques are presented.

2.4.1 Back-EMF

The phenomenon of back-EMF can be used to estimate the velocity of DC motors. As the motor rotates, the coils move through the magnetic field of the permanent magnets, which results in a generated voltage. This induced voltage, or electromotive force, opposes the voltage that is applied to drive the motor. Thus it is called back-EMF.

The faster the motor rotates within the magnetic field created by the permanent magnets in the DC motor, the more back-EMF is induced. This is caused by the fact that the relationship between the back-EMF and the rotational velocity of the motor is directly proportional. Furthermore, if an ideal DC motor with zero resistance and inductance as well as with constant RPM experiences no load, the back-EMF equals the terminal voltage [7].

2.4.2 Current ripples

As opposed to model based approaches based upon, for example back-EMF, ripple based velocity and position estimation relies upon periodic pulses in the supplied current or voltage that are produced as the result of two commutator segment pairs undergoing commutation simultaneously. This "ripple" is directly proportional to the number of commutator segments and to the velocity of the motor. Given such a ripple, the rotational velocity of the motor in rpm can be calculated using Equation 2.4 below, where f_r is the observed ripple frequency in Hz and N_{cs} is the number of commutator segments [5].

$$n_{rot} = \frac{f_r * 60}{N_{cs}} \tag{2.4}$$

If a general brushed DC motor, such as the one illustrated in Figure 2.1, is analysed, current ripples can be found by applying a constant voltage across the motor, and by extension across the brushes. Since the contact point between brushes and commutator segments will be in constant motion relative to the motor axle there will be two possible cases of contact. Either the brushes make contact with just one commutator segment pair or, if the brushes are in the process of switching from one pair of segments to another, two pairs can undergo commutation at the same time. This momentarily changes the total impedance across the motor and gives rise to a periodical pulsation in the supply current. This shift can be observed as the sought after ripple component across the main DC current [6].

The commutation process that corresponds to the the creation of current ripples is illustrated in Figure 2.2 below. In the simplified figure, the motor is made up of an armature with winding and commutator, whereas the commutator has 6 segments and one permanent magnet pole pair. Starting with the model to the left, the brushes makes contact with a single pair of commutator segments. This means the each branch of coils is comprised of three coils each through which current flows. As the rotor continues to move in relation to the stator, the brushes reconnect with the next pair of commutator segments. They do this whilst still being in contact with the previous pair. This is illustrated in the figure to the right. When contact is made with two pairs of commutator segments, the two commutator segments on the same connection side are connected, resulting current flowing through the coils between these two pairs. As a results, each branch of coils will now only comprise of two coils, which means that the total impedance of the motor is lowered momentarily. Since the voltage is kept constant whilst the impedance is lowered, the current will rise. However, since the armature is rotating and the voltage is induced in the coils, the example of impedances is given just to provide a simpler introduction to the topic.

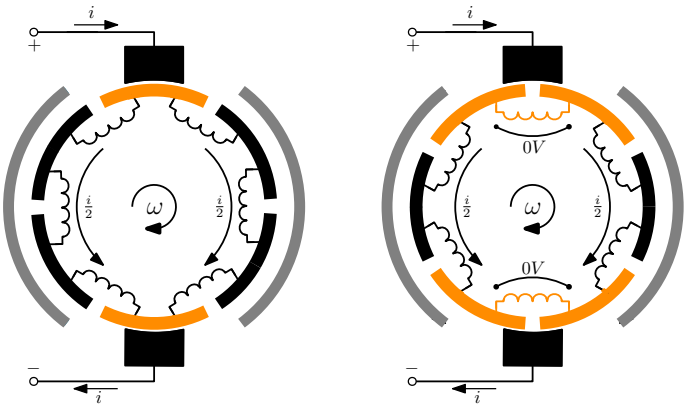

Figure 2.2 When the brushes connect two pairs of commutator segments simultaneously, the impedance of each branch of coils is reduced.

Because of the behavior described above, each coil goes through three stages. First the current flows through the coil towards the opposing brush. Once the coil is disconnected from the neighbouring commutator segment the flow of current suddenly stops, only for the initial current direction to be reversed once the neighbouring commutator segment releases from the brush. These changes, shown later in Figure 3.2, in the current flow gives rise to an inductive behavior which expresses itself as an AC ripple component super-positioned on the larger DC current [8]. However, it should be noted that this explanation is simplified and does not fully capture all the complex interactions in the DC motor.

2.5 Signal handling

When dealing with signal handling, both passive and active filters are valuable tools. This results from the fact that the general measurement signal is a super-position of a large spectrum of frequencies. Some of which are desired and others that are noise. By utilizing filters it is possible to remove frequencies that are undesirable. For example, a low-pass filter will attenuate any frequencies above a certain cutoff frequency, whereas a high pass filter will attenuate any signals under a chosen cutoff frequency.

By also utilizing operational amplifiers in the filter design, so called active filters can be achieved. One large benefit of such a design is the possibility of achieving output gain that surpasses unity, i.e amplified signals. This is in sharp contrast to simple RC filters, where gain will always be under 1. Another useful implementation of operational amplifiers in a filter circuit is that they can be used

as a high impedance step in the circuit which does not load the source. This is especially important if the source signal needs to remain unaltered by the implemented measurement circuit.

By combining different filter circuits it is possible to format the measured signal to an appropriate output. This is important if the source signal voltage and current is too high to be fed in to an analog-to-digital converter of a microcontroller directly. In the following subsections, an overview of the filtering techniques of interest for this thesis will be presented.

2.5.1 The RC low-pass filter

The RC low-pass filter is a filter made up of passive components. As the name suggests, they are resistors (R) and capacitors (C). As discussed in an application from Texas Instruments [9], most filters that operate at frequencies higher than 1 MHz usually also consist of an additional inductor. However, at lower frequencies, i.e 1 Hz - 1 Mhz, the inductor value needs to be exceedingly large, making the components too bulky for most applications.

For this discussed reason, combined with the fact that only relatively low frequencies under 1 kHz are of interest for the applications of this thesis work, the focus will stay on filters that do not utilize inductors. One common filter design that fulfills this criteria is the simple RC filter.

As stated in an article relating to RC filter design published on EEPower [10], the general first order RC filter circuit and its frequency response can be modeled as follows:

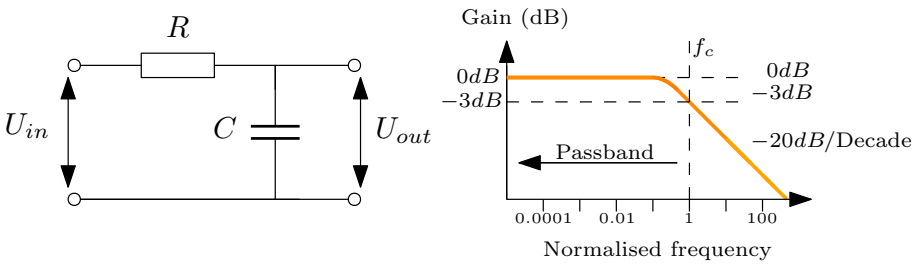

Figure 2.3 A first order RC filter with a corresponding normalized Bode plot.

The cut-off frequency can then be derived from the following formula:

$$f_c = \frac{1}{2\pi RC} \tag{2.5}$$

The simplicity of the RC filter makes it both cheap and reliable. Furthermore, in order to achieve a steeper attenuation of frequencies higher than the cut-off frequency, the design can easily be cascaded as discussed in an article by Electronics

Tutorials [11]. This is done by connecting two first-order filters in series, whereupon a second-order filter is formed. The result is a steeper cut of ratio with the following relationship:

$$k = N * (-20\ dB/Decade) \tag{2.6}$$

where k is the attenuation slope, N is the number of cascaded first-order filters and -20 dB/Decade is the attenuation slope of a single first-order filter.

2.5.2 The active low-pass filter

Since LRC circuits are, as discussed above, unsuitable for the given application of this thesis, active low-pass filters provide additional advantage aside from their amplificational effects. They also provide a means of achieving behavior close to that of LRC circuits for low frequencies.

2.5.3 The Sallen-Key low-pass filter

The second-order Sallen-Key low-pass filter is a relatively easy active filter design to implement. A useful characteristic is that it provides high input impedance and low output impedance. Furthermore, it can easily be cascaded to increase higher order filtering characteristics [12].

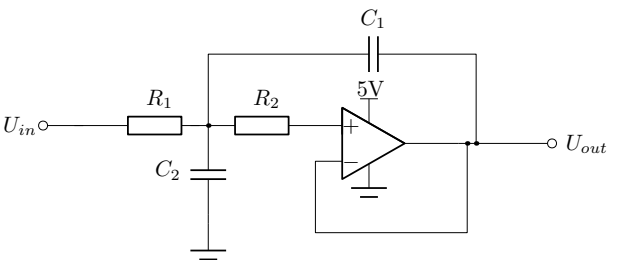

Figure 2.4 The second-order Sallen-Key low-pass filter configured with unity gain.

The fact that the Sallen-Key filter is a popular and easily adjustable filter design means that there is a multitude of design guides available. An example of such is an application from Texas instruments [13] that walks through the design considerations that have to be made when constructing such a filter.

2.5.4 Comparator circuit with hysteresis

An operational amplifier connected as a comparator differentiates between two input signals at the inverting and non-inverting input terminals of the amplifier. It will then drive the output to the positive or negative supply voltage, based upon which input is the largest. This is done by essentially amplifying the difference to such an

extent that any difference between the terminals will lead to a voltage either close to the positive or the negative supply [14].

A common problem with such a on-off comparator is related to noise in the input signals. This is discussed in another application from Texas instruments [15], where it is clear that a noisy signal will lead to additional transitions when the two signals are close to each other. One way to minimize this behavior is by introducing a feedback resistor between the output and the input channel, thus creating a hysteresis. By adjusting the values of the resistors, the span of the hysteresis can be altered as given by Equation 2.7 taken from the lecture notes in the basic course in electrical engineering, EEIF35, given at LTH [16].

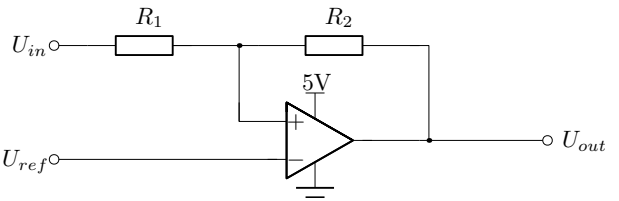

Figure 2.5 A general non-inverting comparator circuit with hysteresis.

$$\frac{U_{\text{in}} - U_+}{R_1} = \frac{U_+ - U_{\text{out}}}{R_2} \Longrightarrow U_+ = \frac{U_{\text{in}} R_2 + U_{\text{out}} R_1}{R_1 + R_2} \tag{2.7}$$

2.5.5 Differential amplifier

As discussed in chapter 6 of Principles and Applications of Electrical Engineering [1], the differential amplifier is another useful implementation of an operational amplifier. Its main purpose is to take the difference between two input voltage signals and output this difference. This means it responds only to the differences between the two inputs, rejecting any voltage common to both. This can be seen in Equation 2.8.

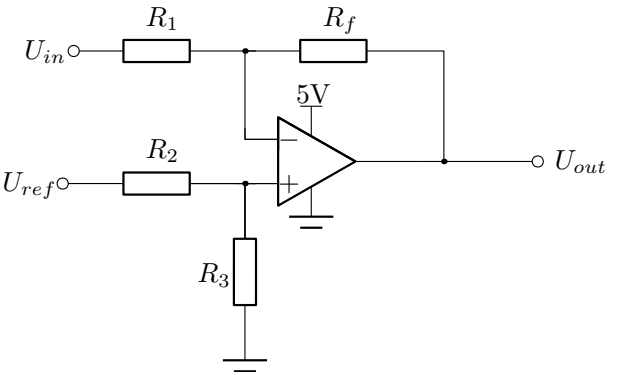

Figure 2.6 The differential amplifier circuit.

$$\frac{R_f}{R_1} = \frac{R_2}{R_3} \implies U_{\text{out}} = \frac{R_f}{R_1}(U_{\text{ref}} - U_{\text{in}}) \tag{2.8}$$

2.6 Current-Control

When it comes to electrical components such as motors, the quantity that is primarily controlled is typically the motor torque, with current serving as an intermediate target. In applications where the mains-grid is connected to power electronic applications via a converter, the converter's purpose becomes injecting or controlling currents. For this reason, current is the focus of most control algorithms. Additionally, for other applications including battery charging and lighting, the primary variable that is regulated is typically the current [3].

A common way to control motor torque through current control is by using an H-bridge. These circuits manage constant current or motor movement. By controlling the current flow, H-bridges ensure that motors operate smoothly, which is important for applications that require precision and reliability.

2.6.1 The H-bridge

The H-Bridge is a circuit which allows a current to flow in two directions. It is often used in motor circuits to enable bi-directional operation. The simplest circuit for the H-bridge is using 4 transistors, as illustrated in Figure 2.7. While Q_1 and Q_3 are open at the same time as Q_2 and Q_4 remains closed, a current flows through the motor forcing it to rotate in one direction. If, instead, Q_2 and Q_4 are opened whilst Q_1 and Q_3 are closed, the current will flow in the reverse direction through the motor, making it rotate in the opposite direction.

It's crucial to ensure that only one pair of diagonal transistors are open at any given time to prevent short circuits. This is the main purpose of utilizing the H-

bridge topology. In general, the transistors used are MOSFETs which offers good reliability as well as power efficiency [17].

Figure 2.7 also illustrated nodes "a" and "b", which marks the points of interest when sampling the voltage "u" across the motor. Furthermore, the switching frequency of the transistors is several tens of kHz, meaning any effect caused by such switching should be negligible in the scope of the ripple frequencies that are of interest in this thesis, which are under 1 kHz.

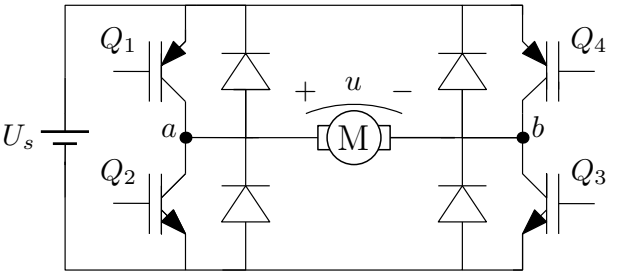

Figure 2.7 A typical H-bridge with four quadrant setup.

2.7 Arduino MKR Zero

Arduino MKR Zero is a development board created by the company Arduino. It uses the Arm® Cortex®-M0 32 bit SAMD21 processor with dedicated SPI interface for communication with external devices and the built-in micro SD card. In Table 2.1, an overview of the technical specifications for the Arduino MKR ZERO is provided. This includes the clock speed, maximum input voltage, pins and more [18].

Table 2.1 Technical Specifications for the MKR ZERO [Arduino.cc., 2024][18].

Category	Details
Pins	
Built-in LED pins	32
Digital I/O pins	8
Analog input pins	7 (ADC 8/10/12 bit)
Analog output pins	1 (DAC 10 bit)
PWM pins	13 (0-8, 10, 12, A3, A4)
External interrupts	10 (0, 1, 4, 5, 6, 7, 8, 9, A1, A2)
Clock Speed	
Processor	48 MHz
RTC	32.768 kHz
Memory	
SAMD21G18A	256 kB Flash, 32 kB SRAM
Power	
I/O voltage	3.3 V
Input voltage (nominal)	5-5.5 V
DC current per I/O pin	7 mA
Supported battery	Li-Po single cell, 3.7 V, 700 mAh minimum
Battery connector	JST PH

2.7.1 Analog pins

As can been seen in Table 2.1, there are 7 analog pins (A0-A6) which can receive analog input voltages. These are converted to a digital value which depends upon the analog bit resolution. One of these input pins can, however, also be configured as an analog output pin which utilizes an onboard digital-to-analog converter, or DAC. As can be seen in the schematic of the pinout shown in Figure 2.8, pin A0 is the digital pin with DAC capabilities. Furthermore, as can be seen in Table 2.1, all input and output voltages received or produced on all pins need to be in the range of 0 to 3.3 V. The only exception being pin V_{In} which uses a positive 5 volt supply.

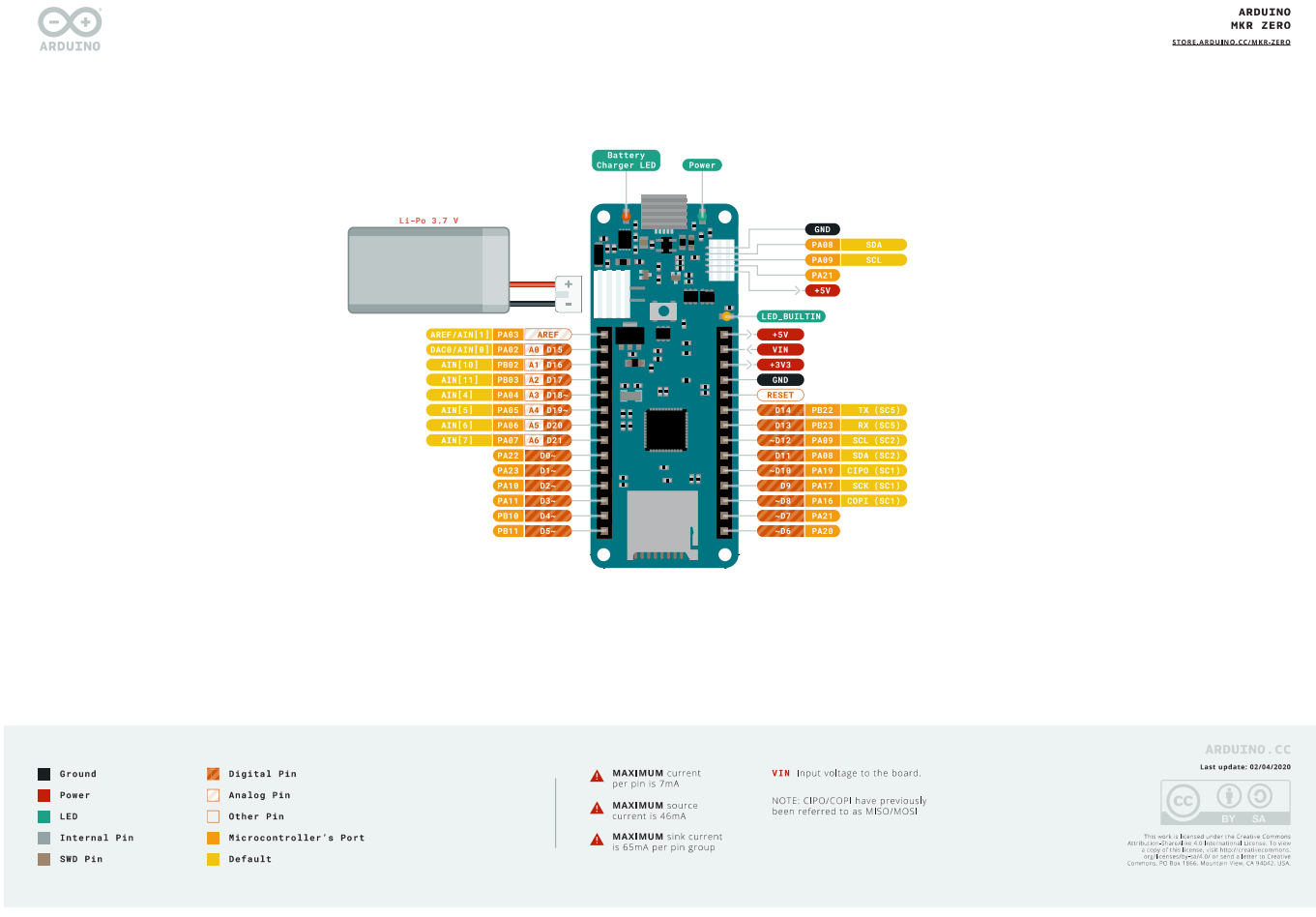

Figure 2.8 An overview of the Arduino MKR Zero pin layout [18].

Analog-to-Digital Converters. As described in the datasheet of the SAMD microcontroller [19] (pages 780-815), these devices include a built-in analog-to-digital converter (ADC) capable of converting analog signals into digital values. The ADC offers a selectable bit resolution ranging from 8 to 12 bits. This resolution is determined by the setting of the result-resolution bit in the Control B register. The ADC is capable of sampling at speeds of up to 350 kS/s at ideal conditions. However, depending on the resolution of the conversion, this speed can vary with the slowest conversion speed produced with a 12 bit resolution. The conversion starts with first selecting a clock which the ADC should be run on. Then, preferences such as bit resolution, averaging, oversampling, hardware gain, offset compensation, and more are configured. Once this has been done the clock needs to be enabled using the ADC clock register where after it has to wait for synchronisation between clocks to finish. The synchronisation is important due to inherent asynchronicity between the main clock domain and the peripheral clock domains which have to be aligned. When this is done the SAMD is ready to read an analog value when called upon.

To read an analog value using the Arduino, a similar principle is used. The Arduino library provides a function called analogRead() using the principle described above. Before reading values using the analogRead(), the bit resolution needs to be set using the function analogReadResolution(), which is used for setting the resolution for the Control B register. Thus determining how finely the ADC can represent an analog voltage. By default the resolution is set to 10 bits but it can be changed with a function call. Before initiating any readings using analogRead(), it's essential to enable the ADC clock using the control A register, which starts when it is set to one. After enabling the clock, data conversion can be initiated by setting the software trigger register to one as well. The ADC is now fully operational and able to find the digital value corresponding to the analog input. To read an analog value, a while-loop is used to wait until the ready flag is triggered. Once the ready flag is set the value from the result-register is stored the control A register enable is set back to zero, turning off the clock. The digital value corresponding to the analog input is then returned by the end of the analogRead()-function after being mapped to the current bit resolution. It is also worth mentioning that the analog value of the Arduino is only able to read positive analog values [20].

Digital-to-Analog Converters. One of the features of the SAMD microcontroller is its built-in digital-to-analog converter (DAC) which can convert digital signals to analog values, which is described in the SAMD datasheet [19] (pages 843-848). As mentioned before there is one pin with DAC capabilities on the Arduino, pin A0, referenced in the SAMD datasheet as PA02. Contrary to the ADC, the resolution is set to a fixed value of 10 bits and cannot be changed. The setup process is similar to that of the ADC with configuring of options such as DMA, multiple trigger sources and more. To use the DAC, a generic clock needs to be selected, configured, and enabled. After synchronization with the main clock is completed, the data stored in the DATA register can be converted to a voltage according to the formula 2.9, where 0x3FF corresponds to 10 bit hexadecimal and the U_{ref} can be chosen by writing to the reference selection bits in the Control B register. After loading the data, the DAC can start the conversion process either automatically when the data is loaded or by a START event from the DATABUF register.

$$U_{\text{in/out}} = \frac{\text{DATA}}{0x3FF \text{ (Resolution)}} \cdot U_{\text{ref}} \tag{2.9}$$

To write an analog value from the Arduino, the Arduino library provides the function anlogwrite(). This function operates differently depending on the pin to which the value should be produced on. If the output is on pin A0, it generates a true DAC (digital-to-analog converters) signal from the SAMD microcontroller. For other pins, the output is in the form of a PWM (Pulse-width modulation) signal. The process for generating a DAC signal starts with the analogWrite() first checking the state of pin A0 to determine whether it is configured as an output or input. It then maps the parameter value passed to the function towards a 10 bit resolution. Shortly after, the DAC clock is enabled, and the conversion process starts automatically after verifying that the value is a 10 bit integer and that it has been added to the data register. If no other writing resolution is set for the Arduino MKR Zero, it defaults to an 8 bit output, which is then converted to the standard 10 bit resolution for the SAMD microcontroller [20].

2.7.2 Digital pins and HIGH|LOW

As shown in Figure 2.8, almost all the pins on an Arduino can be configured as digital pins, offering the flexibility to be used either as input or output for digital signal handling.

If the pins are configured as input pins they have minimal impact on the circuit they are connected to, this is due to their high input impedance of around 1 $M\Omega$. As a result of this, it takes very little current to change the state of the pin. This can lead to problems if nothing is connected to them where, due to electrical noise from the environment, the pin state can exhibit seemingly random changes. When configured as an output pin it is considered to be in a low-impedance state, meaning

that it provides a relatively high amount of current to other circuits of up to 40 mA. This allows it to power circuits that require slightly higher current [21].

The digitalRead() function in the Arduino library is used to check the voltage at a specified input pin and returning its current state. For the Arduino MKR Zero models, this means that it returns HIGH (true) if the voltage is above 2 V, and LOW (false) if the voltage is below approximately 1 V. This function provides an immediate reading of the pins current state, reflecting the current voltage at the input of the digital pin. When a pin is configured as an output, a HIGH state corresponds to a voltage of 3.3 V and a LOW state to approximately 0 V [22].

2.7.3 Interrupt pins and ISR routine

As previously stated in Table 2.1, the Arduino supports attaching external interrupts or ISR routines to 10 different pins. The purpose of an ISR routine is to pause the main routine, handle an external event, and then return to the main routine exactly where it was interrupted. These ISR routines or interrupts can be triggered by multiple different circumstances, such as a voltage change on a pin or a timer overflow. When the ISR is triggered, it is important that it is handled quickly, because lengthy interrupts can affect the code and other important routines for the microcontroller. If the code contains multiple ISR routines and they are triggered simultaneously, the interruptions are queued until the routine with the highest priority finishes. Due to the possibility that ISR routines could change the values of variables at unwanted times, potentially causing concurrency issues, variables shared between ISR functions and normal functions should be declared volatile. This ensures the compiler reloads the variable whenever it is referenced in the code. It is also important to protect certain parts of the code that are critical and should not be interrupted by the ISR routine. This can be done by disabling interrupts before the critical part and enabling them right after [23].

Interrupts in Arduino are enabled by first defining an ISR routine function, which can be triggered by a defined event. This function should not have any parameters or return values, unlike other functions. To attach the newly defined ISR routine to an event, the Arduino library provides the attachInterrupt() function, which requires the cause of the trigger (such as a clock or a pin), a function to handle the interrupt (the ISR routine), and the condition that triggers it, like a change in voltage at the specified pin. The priority of the ISR is connected to the pin number; for example, pin 0 has a higher priority than pin 3 and will interrupt the other if triggered. To protect against this, you can disable interrupts with either noInterrupts() or cli(), and re-enable them with either interrupts() or sei() [23].

2.8 Software methods

2.8.1 Simple moving average and weighted mean

As described in chapter 15 of "The Scientist and Engineer's Guide to Digital Signal Processing" [24], the software solution called a rolling mean, also known as the simple moving average (SMA), is a highly effective method for filtering digital signals from noise. Thanks to the simplicity of the algorithm the SMA can easily remove noise while retaining a sharp step response. This simplicity makes it especially suitable for computer applications, where the SMA can be executed with impressive speed when implemented correctly. The formula for the simple moving average is presented in Equation 2.10. Here, x is the input signal, y is the output signal, and M is the number of points used in the moving average.

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j] \tag{2.10}$$

Unlike the simple moving average, the statistical method of the weighted mean focuses on assigning different weights to specific values in a data collection, rather than treating every value with equal importance. In Equation 2.11 you can see that the sample value x_i is delegated a weight w_i. After combining all the samples with their respective weights, the total value is divided by the sum of the weights [25].

$$y = \frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i} \tag{2.11}$$

2.8.2 Circular buffer

As mentioned in chapter 28 in the book "The Scientist and Engineer's Guide to Digital Signal Processing" [24], a huge problem with digital signal processors is the handling of the real-time problems that comes with it. Ideally the output signal should be produced at the same time as the input signal is received, but this is of course impossible. Real-time applications must instead receive a sample from the environment, perform an algorithm or process, and output the processed result, over-and-over which creates a delay between the output and input signal. However, by working to reduce the delay that results during signal processing the less likely it is to affect the whole process. Oftentimes real-time applications need to rely on both the current and previous samples in order to determine an appropriate output. Doing this facilitates the need of a method that continually updates the memory as new data is acquired. One such solution that offers fast performance is the circular buffer.

The circular buffer works as illustrated in Figure 2.9. The first vector contains 8 samples, where x[n] represents the newest value and x[n-7] represents the oldest. The other vector symbolizes what happens when a new sample is received. As seen in the figure, the oldest value from the previous vector, x[n-7], is replaced with

the newest sample, x[n], and all references move one step ahead, connecting the end of the linear array to its beginning. In short, the circular buffer replaces the oldest sample value in the linear array, and the pointer moves one address ahead, connecting everything in a circular pattern.

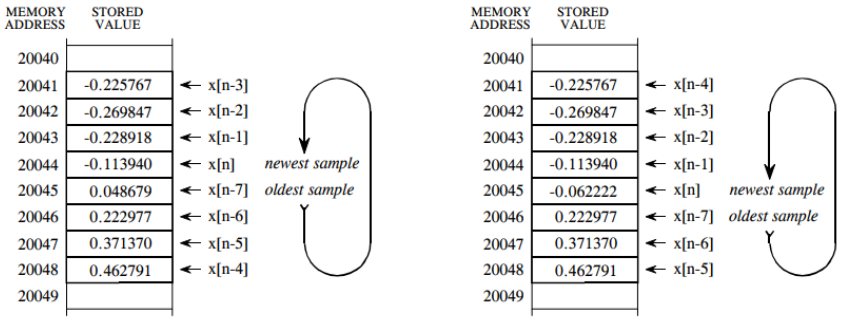

Figure 2.9 A circular buffer [24].

2.8.3 State based programming

A state machine is a model of a system with discrete dynamics, producing outputs in response to specific inputs based on its current state. The finite-state machine (FSM) is a type of state machine with a finite number of states as its name suggest. An FSM begins at an initial state and transitions between states based on conditions known as guards. These guards are boolean-valued expressions that must be true for a transition to be enabled, thereby changing the state from its initial point to its endpoint. It is possible for a transition to start and end in the same state, and during a transition, outputs or actions may be generated [26]. To visualize the already described FSM look at Figure 2.10

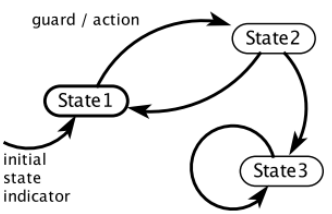

Figure 2.10 The general structure of a finite state machine [26].

3

Experimental work

In this chapter the experimental work of the thesis is presented. By examining and breaking down the task in to smaller sub-problems the work could be carried out incrementally and be divided upon multiple experiment steps.

3.1 Step 1: Current ripples on an ideal setup

The first step in the process was to confirm that current ripples could be detected on a brushed DC motor. This process is explained in the following sub-chapters.

3.1.1 Step 1: Method

As discussed in chapter subsection 2.4.2, current ripples as a phenomenon are the products of the mechanical switching between commutator segments and brushes in brushed DC motors. In order to confirm the presence of such ripples upon a generic DC motor used in many ASSA ABLOY sliding door applications, an ideal test setup was created. By mounting two motors together with each other by coupling their axis together, one motor could be powered by a stable power supply that produces a constant voltage whilst the other one acted as load. The motor specifications are listed in the table below:

Table 3.1 Motor Specifications: Nominal speed (N_n), Nominal voltage (U_n) and Nominal current (I_n).

Unit	Value
Nominal speed (N_n)	3250 rpm
Nominal voltage (U_n)	32 V
Nominal current (I_n)	3.6 A

In order to detect ripples an ammeter was placed in series with the voltage source. The setup is visualized in the diagram below:

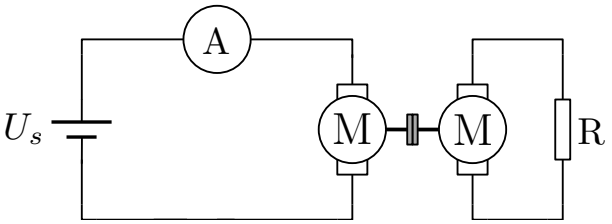

Figure 3.1 The ideal test setup used to detect current ripples.

3.1.2 Step 1: Result

The oscilloscope reading following the experiment described in section 3.1 is presented in Figure 3.2 below. The experiment was only conducted at a single velocity as well as with constant load. The reading was, in turn, produced at a constant voltage of 4.8 V, generating a ripple frequency of $f_1 = 65\ Hz$. Furthermore, the motor shaft could be visually observed to be operating at about $f_r = 5.4\ Hz$, or 324 rpm. This is in line with the expected and generated results, as the expected frequency can be found by multiplying the number of commutator segments of the motor with the shaft velocity in Hz as follows, shown in Equation 3.1 derived from Equation 2.4:

$$f = f_r \cdot N_c = 5.4 \cdot 12 = 65\ Hz \tag{3.1}$$

As can further be seen in the figure, a Fast Fourier Transform (FFT) is superimposed in the window with a centering frequency of 65 Hz, showing that the generated results agree with the expected results. The ripple velocity can also be calculated straight from visual examination of the oscilloscope window, given that it spans 100 ms. Thus, the observed 6.5 ripples can then be divided by 0.1 to yield the resulting ripple frequency. It is worth mentioning that the current was sampled using external current sensing hardware, and that the signal is reproduced as a voltage that is fed to an oscilloscope. Because of this reason the exact amplitude of the current ripples is hard to ascertain. However, these findings still prove that current ripples can be observed using an ideal setup.

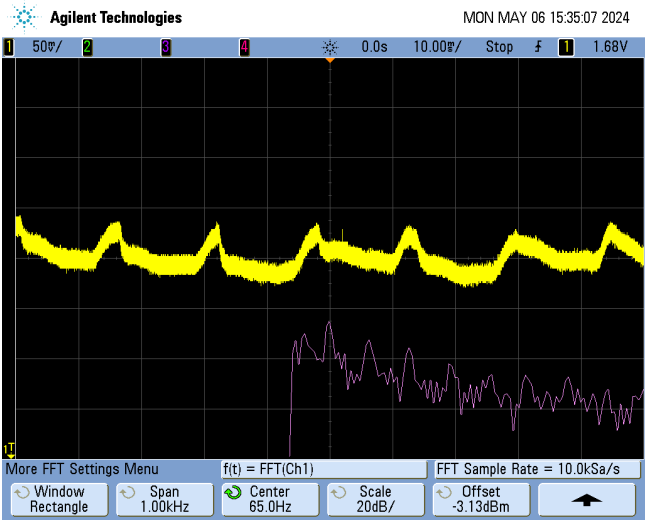

Figure 3.2 The oscilloscope readings produced at 4.8 V and 1 A.

3.2 Step 2: Finding voltage ripples

Given that the existence of current ripples have been proven, the work could move on to examining the possibility of adapting the phenomenon in a format that works for ASSA ABLOY applications.

3.2.1 Step 2: Method

It is discussed in subsection 2.4.2 that current ripples occur in brushed DC motors under the assumption that the voltage is constant. However, the ASSA ABLOY sliding door systems utilizes current control techniques for motor control. This means that any low frequency current ripples are likely to be suppressed by the high frequency current control loop.

However, since current ripples can be observed if the assumption that a continuous voltage is supplied across the motor holds true, the same kind of ripple should be detectable upon the voltage if the current is set to be constant instead. To visualize this, consider Ohms Law. In the previous experiment the voltage U was kept constant while the impedance Z changed during the commutation process resulting in the commutation current ripples. If, instead, the current I is kept constant, the ripples will be forced to express themselves upon the voltage instead.

$$I = \frac{E}{Z} \quad \Longleftrightarrow \quad E = IZ \tag{3.2}$$

If the assumption that the current control loop is fast enough to suppress the current ripples holds true, the motor should in practice experience a constant current

for any constant rotational velocity. Additionally, the assumption can be accepted if the induced voltage is assumed to be constant, as it was not measured or analyzed in this test setup. However, to truly identify the induced voltage behavior during commutation, a more detailed analysis is recommended. Nonetheless, these assumptions is deemed sufficient for this test.

To test this theory, the setup from section 3.1 was modified. The voltage source was exchanged for a constant current supply and the ammeter was exchanged for a voltmeter. In this case an oscilloscope was used instead of a voltmeter to detect ripples with sufficient accuracy. The modified setup is illustrated below.

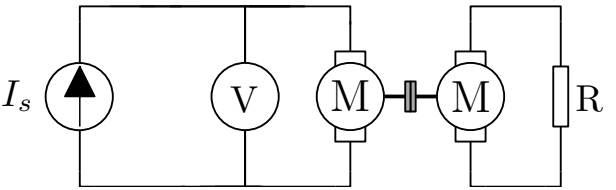

Figure 3.3 The ideal test setup used to detect voltage ripples.

3.2.2 Step 2: Result

After modifying the motor as described in subsection 3.2.1, Figure 3.4 illustrating the oscilloscope reading of the voltage across the motor was produced. The rotational velocity of the motor shaft was 1.25 Hz, or 75 rpm, which was expected to produce a ripple frequency of 15 Hz. This is confirmed by the oscilloscope readings, where manual observation yields a frequency of 15 Hz. Furthermore, the FFT in the window is centered around 15 Hz and indicates that there is a high frequency disturbance at exactly 15 Hz. These findings show that voltage ripples can be found on the given motor models as well.

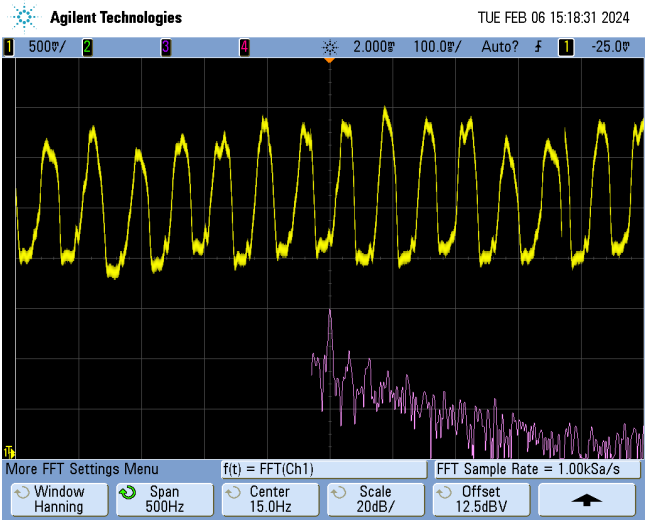

Figure 3.4 The voltage ripples at 15 Hz and an amplitude of roughly 1.5 V.

3.3 Step 3: Finding ripples on a real control unit

With the existence of voltage ripples proven, the logical next step is to examine whether these can be found using the ASSA ABLOY control unit utilizing current control techniques.

3.3.1 Step 3: Method

In order to verify whether voltage ripples could be detected using an actual ASSA ABLOY control unit utilizing current control, the test setup from labs 1 and 2 was extensively modified. The power supply was disconnected and, instead, a fully functioning ASSA ABLOY control unit was connected together with its designated power supply. The mounted encoders of one of the motors was also connected to the control unit in order to provide motor feedback for the integrated software. The same motor test bench was still used however, and the auxiliary motor was still used to provide a simple, latent, load. Since the motors are driven via an internal H-bridge setup on the board, two probes were soldered onto the board at the point of two existing, filtered, measurement signals originating at points "a" and "b" as shown in Figure 2.7. This was done in order to avoid feeding high power signals straight into the measurement equipment as well as examining whether the existing measurement hardware on the control unit could be used to find voltage ripples.

Given this setup, the motor could be manipulated to travel at a constant shaft velocity of 5.4 Hz, or 325 rpm, whilst measurements could be made using an oscil-

loscope. The setup is visualized in the flowchart below:

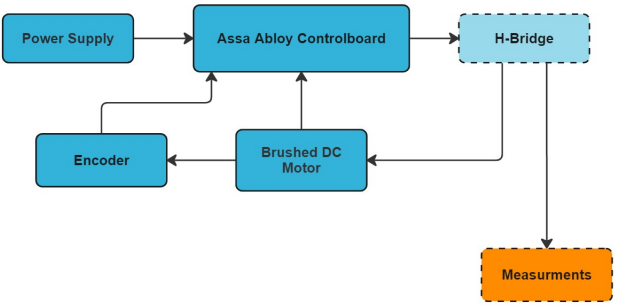

Figure 3.5 An overview of the setup used to identify ripples on the control unit.

One concern that was raised from engineers at ASSA ABLOY regarding utilizing the already existing measurement bridge was the possibility of noise being generated from onboard watchdog sampling at around 1 kHz. Because of this, potential disturbances in the range of 500 - 1500 Hz were investigated using the oscilloscope.

3.3.2 Step 3: Result

Given the above mentioned setup, the initial oscilloscope readings are illustrated in the figures below. Figure 3.6 shows the filtered voltage originating from one side of the H-bridge of the ASSA ABLOY control unit.

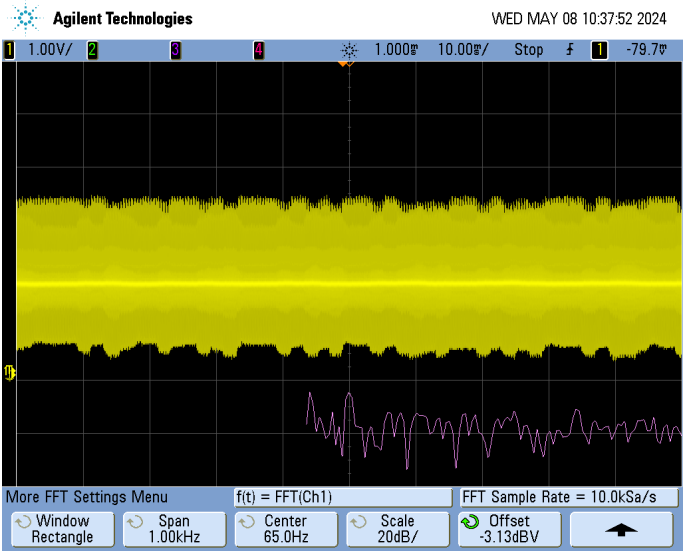

Figure 3.6 The voltage at one leg of the H-bridge as seen after onboard hardware filtering.

In Figure 3.6, a subtle ripple behavior can be observed as spikes with slightly higher amplitude than its surroundings. By counting these through visual inspection it is clear that there is a repeated pattern with a frequency of about 65 Hz. This is confirmed by the Fast Fourier Transform centered in the figure around 65 Hz showcasing a clear spike. The oscilloscopes allows for different intensity settings, enabling the user to perceive different signal intensities as different shades of yellow. This is exploited in Figure 3.7 where the voltage/division has been lowered to provide an enlarged view of the centre of the signal. By doing this a clear commutation ripple trace can be perceived.

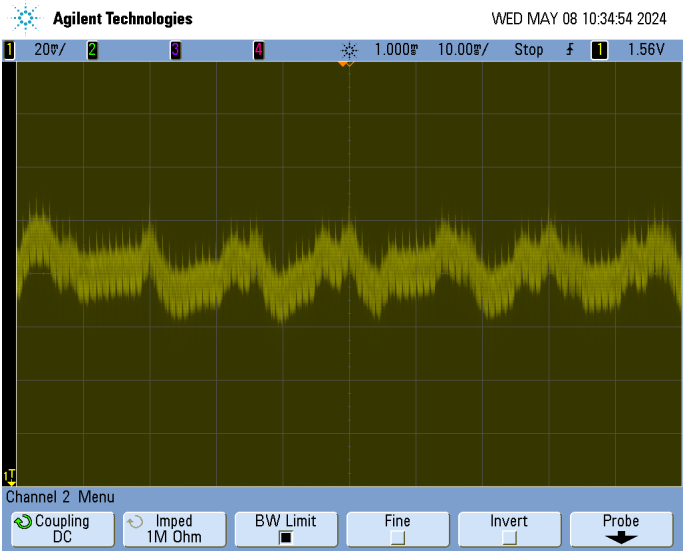

Figure 3.7 The ripple can be seen when tuning down the opacity of high intensity noise.

Investigating the possibility of disturbances caused by watchdog sampling as discussed previously revealed a repeating spike at a frequency of 1 kHz as displayed in Figure 3.8. This is strong evidence of the discussed watchdog sampling proving to be a potential problem. Especially if the signal is to be amplified.

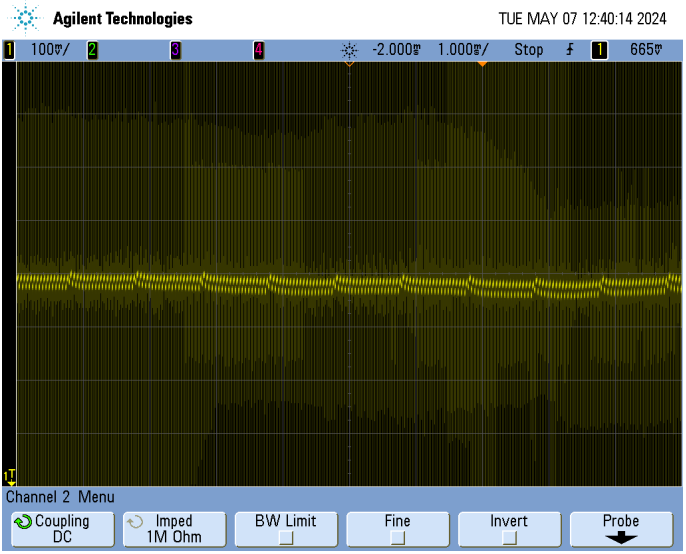

Figure 3.8 Small 1 kHz spikes, likely from watchdog sampling, are observed.

3.4 Step 4: Noise removal and filtering

With the results from the previous experiment, some of the problems could be addressed. In particular removing the watchdog sampling, as well as filtering and amplifying the signal.

3.4.1 Step 4: Method

The first step was to disconnect the measurement signal from the watchdog sampling disturbances seen in the previous paragraph. An attempt at this was done by designing a clone of the current hardware filtering used in experiment 4 and connecting it to one of the legs of the H-bridge. By doing this, the aim is to reproduce the previous signal but without the 1 kHz disturbances.

The second step was to lower the voltage from the newly implemented clone of the analog filter in such a way that it resides within the span of 0-5 V. This is important in order to be able to use subsequent operational amplifiers operating on single rail power of 5 V. Furthermore, in order to be able to test both the onboard filtered connection as well as the newly constructed clone, the input of the filtering solution would have to have sufficiently high impedance. This was done by implementing an operational amplifier as a follower. In order to lower the voltage the amplifier was connected in a high pass filter configuration. As the expected frequencies lies within 0-500 Hz, this was deemed to be an adequate solution for initial testing. Furthermore, in order to provide a clear signal, a low-pass filter was utilized. In this

case, a second-order Sallen-Key low-pass filter was implemented without amplification, and with a cut-off frequency of around 1000 Hz, with the design mentioned in subsection 2.5.3. Together these two filtering solutions provide a filter with band-pass characteristics, and the entire filtering solution is illustrated in Figure 3.9.

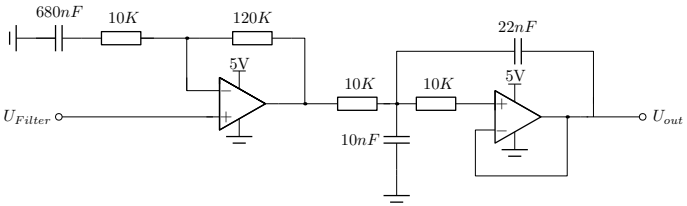

Figure 3.9 An overview over the implemented filters.

3.4.2 Step 4: Result

The results following the above mentioned procedure are displayed in the images below. In Figure 3.10 the yellow signal illustrates the measurements taken from the onboard filtering solution with the watchdog sampling clearly visible at 1 kHz. The green signal shows the cloned solution, clearly indicating that the adverse effects of such sampling has been effectively avoided.

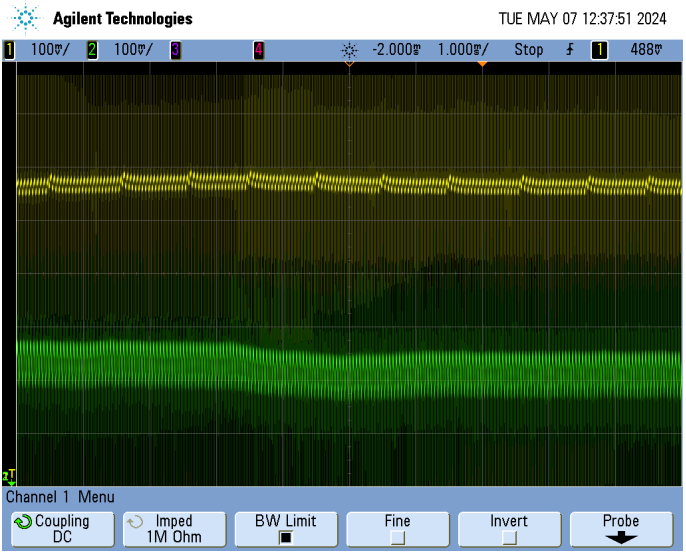

Figure 3.10 The signal with watchdog sampling (yellow) and without it (green).

The signal as seen after the additional high- and low-pass filter solutions is illustrated below in Figure 3.11. The ripple structure is clearly visible, and its frequency

can be counted visually to 65 Hz. As before, the FFT confirms that there is a significant frequency component at this exact frequency.

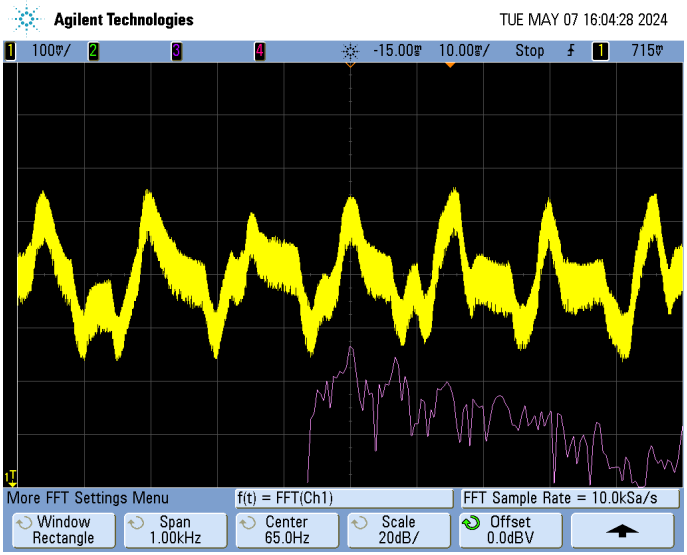

Figure 3.11 The observed ripples after cloning the signal and implementing a band-pass filter.

3.5 Step 5: Finding ripples on a functional test rig

The next step was to examine whether the results in the previous experiment could be reproduced on a fully functioning sliding door. This is an important step to take, as the goal is to examine whether current ripples can be used for velocity and position tracking in real world applications. The methodology and results of this are explained below.

3.5.1 Step 5: Method

In order to verify the functionality of the setup on a complete sliding door system, the brushed DC motor stage used previously would have to be replaced with an actual motor mounted on a beam. The new motor is attached to a belt drive that in turn drives a door leaf. The door leaf was loaded with weights equivalent to 86 kg. The same control unit used in the previous lab was kept but connected to the new peripherals. As such the same probes used to measure the motor signals in experiment 3 could be used in experiment 4 without additional alterations. The changes made to the system is displayed schematically below:

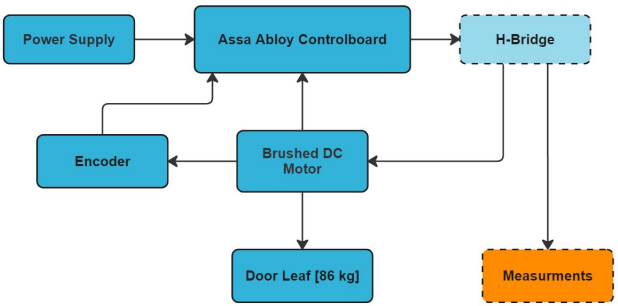

Figure 3.12 An overview of the setup used to identify ripples on a full test rig.

By connecting an external programmer to the system, parameters such as velocity and high/low speed zones could be tuned. This is important in regards to understanding how the ripples behave at different velocities. For the tests conducted in this experiment, an opening velocity of 300 mm/s and closing velocity of 150 mm/s was chosen. This should produce a ripple frequency of 200 Hz and 100 Hz respectively. This would, in turn, correspond to rotational velocities of 1000 rpm and 500 rpm.

3.5.2 Step 5: Result

The results produced in the experiment are displayed in the figures below. In Figure 3.13 an entire cycle is displayed, including an opening sequence spanning between timestamps 2.6 to 11.6 seconds, followed by 2 seconds at standstill after which the closing sequence lies within 13.6 to 29.2 seconds. Figure 3.14 and Figure 3.15 provides an enlarged view of the ripples at opening and closing sequence.

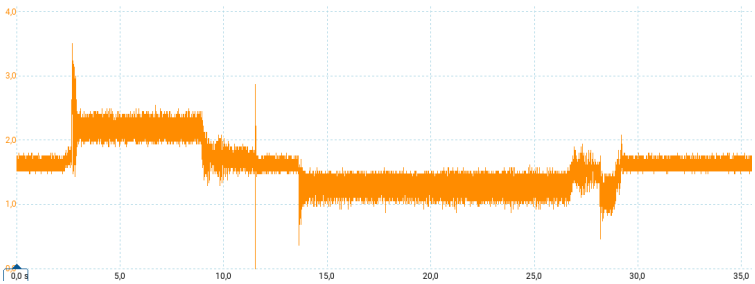

Figure 3.13 A full cycle illustrating opening-, standstill and closing sequences.

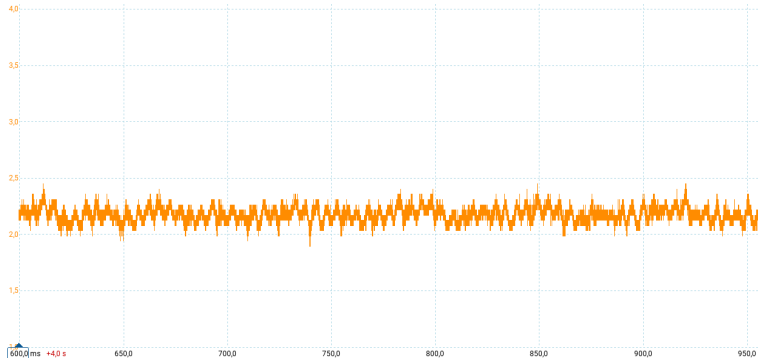

Figure 3.14 An enlarged view of the ripples produced during the opening sequence.

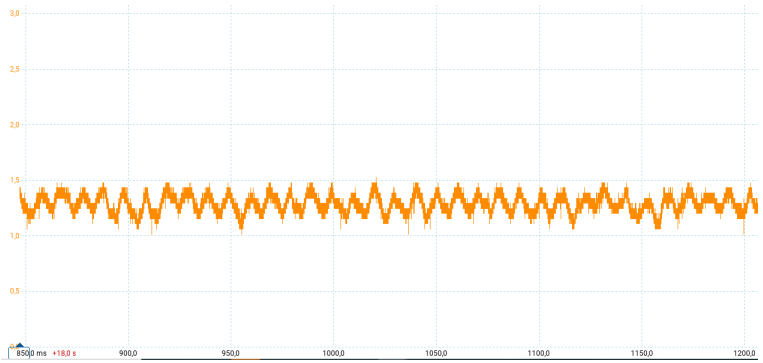

Figure 3.15 An enlarged view of the ripples produced during the closing sequence.

These results clearly indicate that the sought after ripples is detectable at the correct frequencies on the fully functioning door as well. As can be seen in Figure 3.14 and Figure 3.15 there are slight differences in the ripple quality, where the opening sequence showcases a less pronounced ripple overlaid on low frequency disturbances. The closing sequence produced a more distinct ripple with less disturbances, which is a reasonable result given the fact that the sequence more closely resembles the low velocity and constant operation of the motor setup used in the previous experiment.

Furthermore, there are added dynamics that did not exist in experiment 4 brought on by exchanging constant operation for an actual opening and closing cycle. This is for example seen in the added signal noise during acceleration and breaking, and in the differing opening and closing velocity voltages. The graphs also clearly show that there is still a substantial degree of noise in the signal as well, although the ripple component is still clearly visible.

The offset voltages that appear at different velocities is demonstrated more clearly in Figure 3.16 below, where the signal has gone through a low-pass filter function on the picoscope.

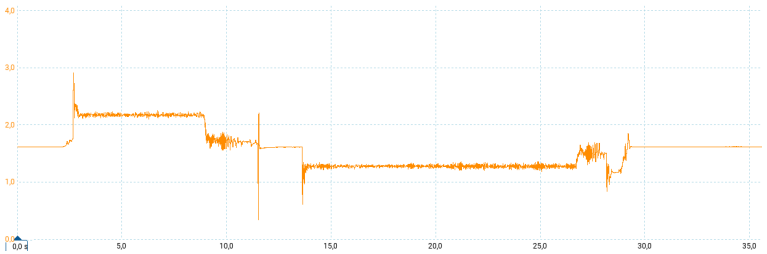

Figure 3.16 The voltage offset as seen using a software low-pass filter function.

As seen in the figure, during the opening sequence the voltage offset climbs from 1.6 V to 2.2 V before returning to 1.6 V at standstill. When the door enters the closing cycle the voltage decreases to 1.28 V before returning to 1.6 V. Furthermore, the figure also clearly demonstrates how the offset voltage decreases and is subject to large amounts of disturbances during the breaking phases.

3.6 Step 6: Problem identification and ADC

Given the results produced in the previous experiment, a number of key areas in need of improvement can be identified. One of them is the problem relating to the large differences in offset voltage relating to operation at different velocities, and the fact that the ripple component remains relatively small in relation to this offset. In order to be able to further amplify this signal, the different offsets have to be removed without adversely affecting the ripple component. This experiment explores the specifications that such a solution needs to meet.

3.6.1 Step 6: Method

As one of the aims of this project is to be able to count the ripples using a microcontroller, the requirements of such a solution have to be examined. The particular microcontroller of interest for this thesis, the Arduino MKR Zero has, as discussed in section 2.7.1, some limitations. These limitations include the ADC only being able to handle positive voltages with a maximum input voltage of 3.3 V. Any signal that exceeds this maximum voltage capacity risks damaging the board or the analog pins. Because of this fact, the measurement voltage being fed in to the Arduino would have to be constrained to voltages under 3.3 V. This requirement was not negotiable and would therefore have to be taken into consideration during all subsequent design steps.

Furthermore, the resolution of the Arduino ADC would have to be taken in to account. This means that the signal ripple could not be too small in relation to the Arduino ADC span in order to avoid quantization errors as well as preventing external noise from polluting the results. With resolutions at 12 bits being 4 times more accurate then 10 bits and 16 times more accurate then 8 bits as given by Equation 3.3 which expands upon Equation 2.9.

$$\frac{12 \text{ bits}}{8 \text{ bits}} = \frac{12 \text{ bit Resolution}}{8 \text{ bit Resolution}} = 16 \tag{3.3}$$

This implies that the chosen resolution is of great importance for accurately measuring the voltage. For example, a resolution of 8 bits would, given an ADC range of 0 - 3.3 V, provide a resolution of 0.01289 V, or roughly 12.9 mV. This can be put in to contrast by increasing the resolution to 12 bits as this would increase the resolution to 0.81 mV. This is also clear from looking at Equation 3.4 and Equation 3.6.

$$\text{8 bit ADC: } 2^8 = 256 \text{ levels, } \Delta V_{8\text{ bit}} = \frac{3.3 \text{ V}}{256} \approx 0.01289 \text{ V} \tag{3.4}$$

$$\text{10 bit ADC: } 2^{10} = 1024 \text{ levels, } \Delta V_{10\text{ bit}} = \frac{3.3 \text{ V}}{1024} \approx 0.00322 \text{ V} \tag{3.5}$$

$$\text{12 bit ADC: } 2^{12} = 4096 \text{ levels, } \Delta V_{12\text{ bit}} = \frac{3.3 \text{ V}}{4096} \approx 0.00081 \text{ V} \tag{3.6}$$

Another important aspect to consider in this context that affects the choice of resolution is discussed in subsection 2.7.1, where it is stated that the time taken for a single analog reading to be taken is dependent upon the resolution. As such a higher resolution leads to a higher degree of processor utilization and might result in slower performance of other tasks. This means that the choice of resolution will have to be made in consideration of total utilization time and processor availability.

In order to verify the choice of resolution, the sampling time of the ADC will be measured at the different resolutions. This will yield an overview of the possible sample frequencies and processor utilization degree that will shape any future work.

3.6.2 Step 6: Result

Given equations 3.4, 3.5 and 3.6 it is clear that the resolution is greatly affected by the number of bits used. Measuring the time required for the Arduino MKR Zero to complete an ADC reading at 8, 10 and 12 bit resolutions yielded the results in Table 3.2 . The column "Maximum Frequency" denotes the maximum achievable frequency given the sampling measured sampling times, and providing that no additional tasks are allocated to the processor.

These results show that there is no significant differences in sampling time between the different resolutions. They do, however, show that the sampling times are

Table 3.2 The measured sampling times and maximum achievable frequency for different ADC resolutions using the Arduino MKR ZERO.

Resolution	Sampling Time	Maximum Frequency
8 Bits	814 μs	1228 Hz
10 Bits	835 μs	1198 Hz
12 Bits	857 μs	1167 Hz

lengthy at 0.800 ms to 0.860 ms. This means that there might be problems associated with sampling fast enough in the future, as more code will have to be integrated outside of ADC sampling. The maximum ripple frequency to be counted is around 460 Hz, this means that during the period of one such ripple, the ADC can sample less than 2.67 times. With a 12 bit resolution this shrinks to 2.54 times. This indicates that there might be problems associated with capturing larger ripple velocities in the future. Since the sample time difference between the 8 and 12 bit resolutions are about 5% it can be concluded that the 12 bit resolution can be utilized without compromising the results.

In order to provide an accurate reading, the amplitude of the ripple will still have to be magnified. Figure 3.14 and Figure 3.15 displays a ripple amplitude of roughly 0.5 V, indicating that the signal could be magnified approximately 5 times, to about 2.5 V without compromising the ADC voltage restrictions of 3.3 V. Doing such would, given a resolution of 12 bits as indicated by Equation 3.6 provide a resolution of 3086 points for the entire amplitude of the ripple. These results show that by removing the offset, amplifying the signal and choosing the 12 bit resolution, the voltage ripples should remain detectable with a high degree of accuracy at lower frequencies.

3.7 Step 7: Removal of the DC offset

As mentioned before, the offset voltage will have to be removed in order for the signal to be amplified. This experiment explores some of the possible approaches to this problem.

3.7.1 Step 7: Method

In order to be able to retain as much information as possible in the signal, an analog solution for offset removal is desirable. One of the ways this could be done is by creating a filtered copy of the signal, where the ripple component is removed, and feeding both the source signal and the filtered clone, or follower signal, to a differential amplifier circuit. As such, the common mode voltage should be removed from the signal, leaving the ripple component without offset. Creating this follower signal can be done in many ways, but one of them is by probing the signal as it appears after the Sallen-Key filter and utilizing a low-pass filter in order to remove

the ripple component from it. A significant problem that arises when utilizing this method is the fact that the ripple frequencies that are to be measured typically resides within a span of 0-500 Hz. In terms of low-pass filters this is extremely low, and even if a filter with a cut-off frequency of 1 Hz is designed it will be hard to ensure a steep enough dampening so as to not affect the ripples, whilst still providing a filter that moves fast enough to capture the rapid changes of the offset voltage during acceleration and deceleration.

Another way would be to sample the signal at the same location just mentioned, but instead by using the ADC of the microcontroller. As such, the signal could be filtered in software and then reproduced on the digital to analog converter pin, or DAC pin, of the Arduino. Given proper filtering, this DAC signal could be fed straight to the differential amplifier, where the offset would be removed. By doing this, the differential amplifier could also be connected in such a way as to provide a gain of 5 times the input signal to meet the requirements described in experiment 6. In order for this to work, another problem would have to be addressed, namely the fact that the differential amplifier is utilizing a single-rail power supply of 5 V only. This means that feeding the DAC signal to the differential amplifier together with the source signal would effectively remove half of the signal, as the output would be centered around 0 V.

3.7.2 Step 7: Result

By creating a differential amplifier as described in Figure 2.6 with all resistors R set to 10 $k\Omega$, U_{in} serves as the input signal from the filtered bridge, and U_{ref} as the input from the Arduino, which only outputs a DC offset. The resulting signal can be observed in Figure 3.17. As shown, the center value is approximately zero, effectively eliminating half of the ripple. Figure 3.18 illustrates the effects of adding a slight offset of around 1 V, equivalent to a value of 1240 for the standard 12 bit output resolution of the Arduino MKR Zero. Thus, handling the offset captures the entire ripple, making it more readable. As demonstrated by the differences between Figures 3.13 and 3.18, the removal of the unwanted offset and the increase in amplitude makes the ripple not only easily identifiable but also more uniform, thereby simplifying the processing for future stages.

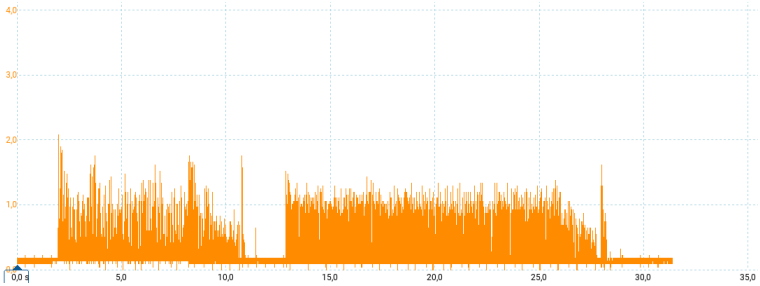

Figure 3.17 The signal as seen after the differential amplifier.

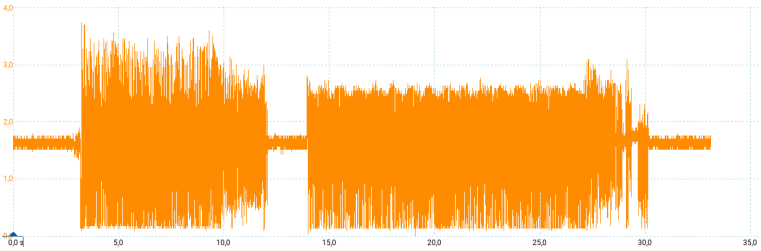

Figure 3.18 The ripple as seen after the differential amplifier with offset added to the input.

3.8 Step 8: Comparator

In order to accurately identify the ripple spikes, they must first be converted in to a uniform structure so that they can be easily read and processed by the given microcontroller. The simplest way to achieve this is to use an analog comparator that transforms the ripple in to a square wave, as dictated by a moving reference.

3.8.1 Step 8: Method

With the ripple identifiable using a picoscope, the next step was to make sure the chosen Arduino microcontroller could read and identify the ripple in order to accurately count them. Doing this necessitates a fast and simple solution that detects the ripple and then structures it in to a uniform waveform that is easily detectable for a microcontroller. The solution to this problem was to utilize a single rail, analog comparator with a reference signal of 1.5 V. The decision to use an analog comparator for the comparator was due to the accuracy of the analog resolution and reading speed compared to that of the ADC and subsequent, comparable, software solution. In order to provide a simple to view graph on the picoscope and to stick with the theme of previous OP amplifier solutions, a non-inverting comparator was used, equal to the one mentioned in Figure 2.5 but without the positive feedback.

3.8.2 Step 8: Result

The resulting comparator signal can be seen in Figure 3.19 below, which displays a grey square wave representing the comparator output, alternating between the rail-to-rail voltages of 0-3.3 V. The orange line is, in turn, the analog signal originating from the differential amplifier and contains the sought after voltage ripple. The blue line in the middle is pre-set reference voltage of 1.5 V. As can be seen in the figure, the square wave is relatively accurate in identifying the sought after ripples. There are, however, some areas where the ripples are read multiple times leading to a more optimistic velocity estimation than necessary.

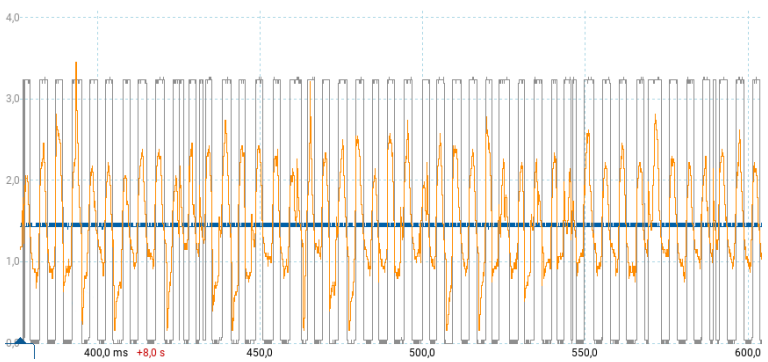

Figure 3.19 The resulting comparator output given a constant reference.

3.9 Step 9: Floating comparator reference

The fixed reference of the previous lab produced some problems that had to be fixed. One of these problems was the fact that some of the ripples were not properly counted as the ripple suddenly floated too far above or below the reference. Thus, a moving reference needed to be implemented and fed to the comparator. Furthermore, the problem of some ripple signatures being read multiple times would have to be addressed.

3.9.1 Step 9: Method

During picoscope measurements of a run, it could be observed that a fixed reference was unreliable in capturing all ripples due to the center of the ripple varying in amplitude. The solution to this problem was to feed the comparator with a varying reference voltage produced by the Arduino. This was accomplished by reading the output from the differential amplifier with the ADC aboard a second Arduino and applying a moving average of 8 samples to this signal, sampled at a frequency of 1 kHz. This averaged signal was then fed to the comparator as the reference value. The function handling the moving average is a generic function shown in the Listing 3.1. It uses the circular buffer structure, as mentioned in subsection 2.8.2, to more efficiently process new samples.

```
/* This function implements a generic rolling average calculation.
 *   -‘mean‘ is an array that stores previously read values.
 *   - ‘readValue‘ is the most recent value to be included in the average.
 *   - ‘index‘ specifies the current position within the ‘mean‘ array.
 *   - ‘totalVectorValue‘ represents the cumulative sum of the values in the ‘mean‘ array.
 *   - ‘sampleNbrBit‘ defines the number of samples, affecting the size of the ‘mean‘ array and the scaling of the average.
 */
int rollingMeanGeneric(int *mean, volatile int *readValue, int *index, int *totalVectorValue, int sampleNbrBit)
{
  // Reduce the total value of the stored vector with the oldest value.
  *totalVectorValue -= mean[*index];
  // Replace the oldest value with newly read value.
  mean[*index] = *readValue;
  // Add the new value to the total value of the stored vector
  *totalVectorValue += mean[*index];
  // Increment the index with wrap-around:
  *index = (*index + 1) % (1 << sampleNbrBit);
  // Divide the total with the sample size
  int RollingMean = (*totalVectorValue) >> sampleNbrBit;

  return RollingMean;
}
```

Listing 3.1 An overview of rollingMeanGeneric() in the class ComparatorRefHandler

Furthermore, in order to avoid the problem of having some ripples being counted multiple times, a hysteresis was implemented, as displayed in Figure 3.19. To do this, a positive feedback loop with R_1 set to 1 $k\Omega$ and R_2 set to 8 $k\Omega$ was used. Equations 3.7 and 3.8 derived from Equation 2.7, shows us that this solution will work in most cases. It is, however, not optimal as using a non-inverting comparator in this manner produces a hysteresis with moving saturation values. With Equation 3.8 highlighting this as there is a constant voltage whose impact increases at lower voltages. In the following equation U_+ represents the voltage going into the positive leg of the comparator, U_{in} is the ripple signal, and U_{out} is the output signal.

$$U_+ = \frac{U_{\text{In}}R_2 + U_{\text{out}}R_1}{R_1 + R_2} = \frac{U_{\text{In}} \cdot 8000 + 0 \cdot 1000}{1000 + 8000} = U_{\text{In}} \cdot 0.89 \tag{3.7}$$

$$U_+ = \frac{U_{\text{In}}R_2 + U_{\text{out}}R_1}{R_1 + R_2} = \frac{U_{\text{In}} \cdot 8000 + 3.3 \cdot 1000}{1000 + 8000} = U_{\text{In}} \cdot 0.89 + 0.367 \tag{3.8}$$

Although this solution is not perfect, it is deemed good enough for the time being, as it requires less time being spent on remodelling the circuits. For future work it might be wise the revisit the comparator design however.

3.9.2 Step 9: Result

The resulting comparator signal given the improvements made can be seen in Figure 3.20, with an orange line representing the analog signal from the differential amplifier. The blue line in the middle indicates a moving average of 8 samples, as outputted from the Arduino's DAC. The yellow square wave represents the comparator output. The moving reference is clearly helping in capturing ripples where the average amplitude suddenly shrinks. Another results of this implementation is the possibility to alter this behavior in software to better tailor the solution for even better results in the future.

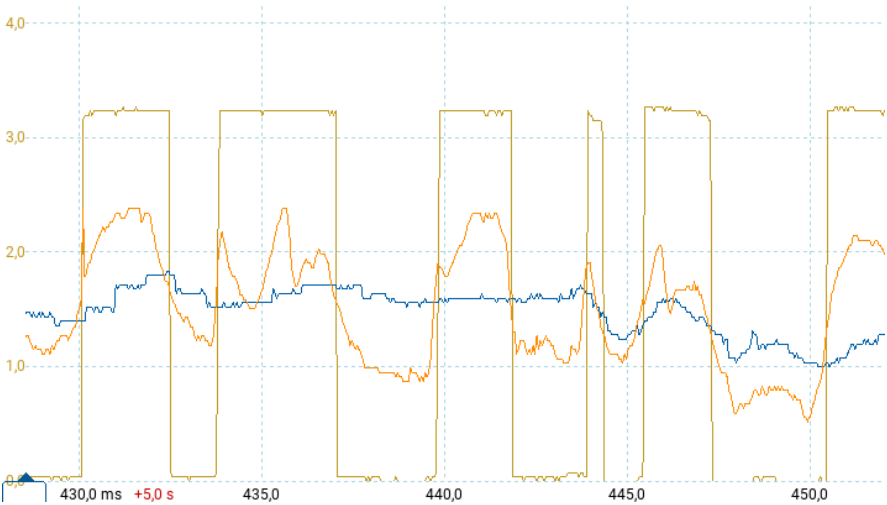

Figure 3.20 The resulting comparator output using a moving reference.

To visualize the implemented hysteresis band, Figure 3.21 was created with two purple lines added post hoc in the picoscope software using the formulated equations 3.7 and 3.8. This shows that the hysteresis provided better results and made the comparator less sensitive to false readings.

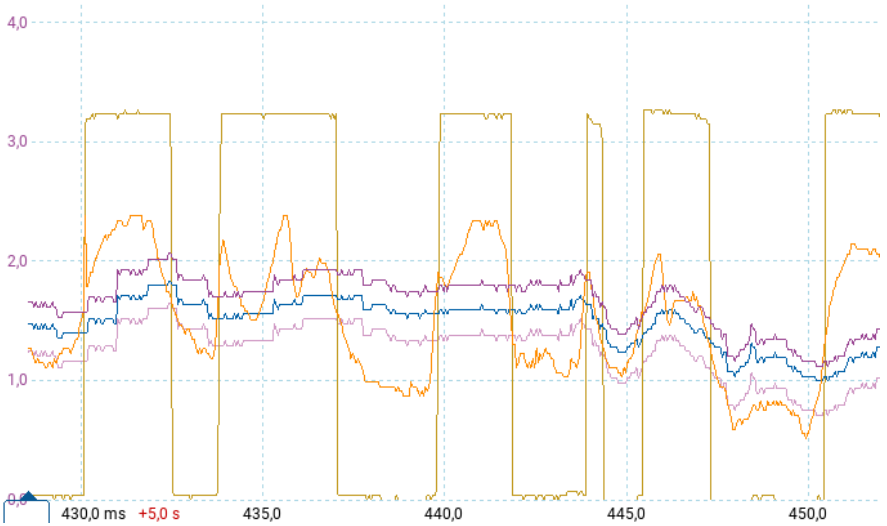

Figure 3.21 The resulting comparator output with the hysteresis visualized.

3.10 Step 10: Ripple analysis and echo rejection

With the comparator implemented and the ripple transformed to a square wave that could be fed to a digital input of the Arduino, the next step was to produce the necessary software required to accurately count the ripples. Even though the implemented hysteresis produced a better results with many false ripples rejected, the square wave was still optimistic in counting the ripples. This meant that software that could filter out unwanted readings caused by repeated spikes of the same ripple and other noise would have to be implemented. The technique is henceforth called "echo rejection".

3.10.1 Step 10: Method

With the signal transformed in to a square wave, it can easily be fed into one of the Arduino's digital pins. This means that an interrupt routine, triggering at rising edges, can be attached to the pin in order to count the ripples. The interrupt routine will essentially be activated every time the comparator output switches from LOW to HIGH, with the interrupt routine software incrementing the ripple counter each time it does so. Once the comparator's outputs could be counted using interrupts, the frequency of the ripple was calculated by counting the number of interrupts over a fixed period of time. Furthermore, the ripple counter was saved periodically along with a timestamp in order for easier post-processing of the data to be conducted.

In this post-processing of the ripple counter and velocity, or frequency, of the door, special care was taken to ensure that unexpected or odd readings would be

handled appropriately as to not cause spikes in the graphs. This was important in order to quickly asses the overall performance of the software solution. The post-processing filter was implemented constructing a moving average filter taking in the last 10 ripple measurements, which helped smooth out the data. This approach removes unusually high or low readings, making the overall results more reliable and providing a clearer picture of the door's movement.

In order to provide a benchmark for the calculated velocity, the onboard motor encoder was utilized in order to produce a more accurate velocity graph. This real time graph will accompany the velocity graphs henceforth and can be used for easy comparison between the ripple solution and existing encoder solutions. It is, however, worth mentioning that, because of the fact that the encoder pulses are registered in parallel aboard the Arduino, the encoder velocity graph may not be as accurate as it could have been. This is caused by the fact that the readings made by the Arduino are more prone to noise than the existing solution on the control unit. Furthermore, some additional noise can be expected during communications between the Arduino and the Data streamer function on excel used to produce the graphs.

As discussed before, due to the structure of the ripple, sometimes comprised of two or three peaks, the comparator's counting was often too optimistic, resulting in a higher-than-expected frequency. This was counteracted by implementing an echo rejection technique that is called upon each time an interrupt is thrown. The echo rejection works by utilizing a set delay time during which additional readings are rejected. The delay is calculated based on the period of a voltage ripple at a certain door velocity. This system should remove the majority of the false readings given by the comparator. There are, nonetheless, some drawbacks. First of all, utilizing this echo rejection delay technique means there is a risk of the system turning into a self-fulfilling prophecy. Furthermore, if a ripple wave is not properly detected or many false readings are produced in series, a phase shift in the readings might occur. However, the expectation is that as long as the majority of readings are correct, such a phase shift will not have any long lasting implications. With this said, the echo rejection technique should still provide more merits than flaws to the overall solution.

For the tests conducted in this experiment, an opening velocity of 300 mm/s and a closing velocity of 150 mm/s were chosen, along with an average of 8 samples for the reference to the comparator. This should produce ripple frequencies of 200 Hz and 100 Hz, respectively.

3.10.2 Step 10: Result

The result of the implemented counting structure, utilizing a 500 μs delay, can be seen in Figure 3.22 showcasing an excel graph over the data gathered from the Arduino. As can be seen in the graph, running both the opening and closing cycles takes around 24 s. It should also be noted that the calculated ripple is more accurate around 200 Hz as opposed to 100 Hz, where the readings are optimistic.

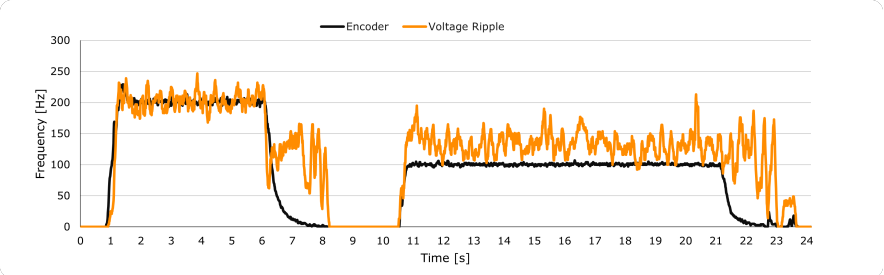

Figure 3.22 The results using echo rejection with a 500 μs delay.

Figure 3.23 displays the calculated velocity with a higher delay of 1500 μs . In this graph, the calculated velocity is instead more accurate around 100 Hz, whereas the counted ripple at 200 Hz is worse than before.

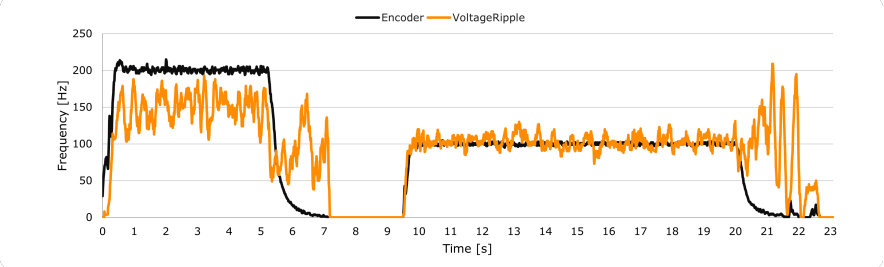

Figure 3.23 The results using echo rejection with a 1500 μs delay.

3.11 Step 11: Echo rejection adjustment

After implementing echo rejection for a single frequency, a state-based structure was designed to handle multiple frequencies, changing states based on the door's velocities. By adding more states, incorporating a slow moving weighted mean between transitions, and reducing the frequency of echo rejection calculations, a more accurate signal was achieved.

3.11.1 Step 11: Method

With the implementation of the echo rejection, the results were much better. However, as seen before, each set delay time corresponded with good results at a very narrow frequency band or state. In order to improve the results over a wider frequency band, more states or frequency ranges needed to be implemented to handle changing velocities. A transition condition was therefore necessary to facilitate movement between frequencies, leading to the use of back-EMF as the transition mechanism. As the back-EMF is proportional to the velocity of the door, it could be used to approximate this velocity in order to dictate when to move between states. In short, a higher back-EMF value would indicate a larger door velocity, and a lower back-EMF indicates a lower velocity. The back-EMF can be derived from the equation for the motor armature voltage described in Equation 2.1. Using this relationship, we can further derive Equation 3.9. In the scenario where the current remains constant, the derivative part becomes zero, as shown in Equation 3.10. To further enhance the back-EMF readings, a mean of four readings was created to smooth the signal. The variables in the equations describe the back electromotive force E_b in an electric motor, where ω_r is the rotor's angular velocity, Ψ_m is the magnetic flux linkage, u_a is the armature voltage, R_a is the armature resistance, i_a is the armature current, L_a is the armature inductance, and $\frac{di_a}{dt}$ is the rate of change of armature current over time.

$$E_b = \omega_r \cdot \Psi_m = u_a - R_a \cdot i_a - L_a \cdot \frac{di_a}{dt} \tag{3.9}$$

$$E_b = u_a - R_a \cdot i_a - L_a \cdot 0 \tag{3.10}$$

During an observation of the back-EMF graph, the corresponding states to different velocity ranges could be identified, thus providing multiple velocity states. Once these states were identified, appropriate delays were set based on the observed ripple duration at different velocities, ensuring more precise control. To further smooth the echo rejection delay transitions between the different states, a mapping was implemented to make the nonlinear transition smoother by breaking it into smaller linear segments. Because the echo rejection value was prone to causing jitter due to frequent state shifts, the echo rejection calculation rate has been fixed at 50 ms, instead of being recalculated each run. Additionally, the transitions should be made less prone to jitter by adding a weighted mean. A weighted mean approach, rather than a rolling mean, was used because switching between mappings caused errors with the rolling mean, but not with the weighted approach. In this case, the weights are 60 % for the last value and 40 % for the new one. This state structure is shown in the Listing 3.2.

```
void adjustableEchoRejectionStateMachine()
{
  unsigned long timeDifference = millis() -
    echoRejectionTime;
  if (timeDifference >= 50)
  {
    // Determine the current state
    int currentState = getBackEMFState(
    UMotorBackEMFValue_RollingMean);
    int maxEchoRejectionMicros=0, minEchoRejectionMicros=0,
    minValueFromBackEMF=0, maxValueFromBackEMF=0;
    // Use a switch-case to handle different states
    switch (currentState)
    {
    case HIGH_BACK_EMF:
     //Set maxEchoRejectionMicros, minEchoRejectionMicros,
              minValueFromBackEMF, maxValueFromBackEMF
      break;
    case MEDIUM_BACK_EMF:
      //Set maxEchoRejectionMicros, minEchoRejectionMicros,
               minValueFromBackEMF, maxValueFromBackEMF
      break;
    case LOW_BACK_EMF:
      //Set maxEchoRejectionMicros, minEchoRejectionMicros,
               minValueFromBackEMF, maxValueFromBackEMF
      break;
    }
    // The map function uses linear interpolation to find
    corresponding EchoRejection time according the back EMF
     speed
    echoRejection_CurrentReadValue = map(
    UMotorBackEMFValue_RollingMean, minValueFromBackEMF,
    maxValueFromBackEMF, maxEchoRejectionMicros,
    minEchoRejectionMicros);
    // To smoothen the echoRejection delay an weighted mean
    with focus on the old value is used.
    echoRejectionDelay = weightedAverage(
    echoRejection_LastReadValue,
    echoRejection_CurrentReadValue, weightLast, weightNew);
    echoRejection_LastReadValue =
    echoRejection_CurrentReadValue;
    echoRejectionTime = millis();
  }
}
```

Listing 3.2 The echo rejection state machine in the class ComparatorRefHandler.

3.11.2 Step 11: Result

To identify the different transition conditions for transitioning between states(echo rejection delays), a run was conducted at an opening velocity of 700 mm/s and a closing velocity of 100 mm/s, capturing the whole spectrum of velocities tested. The back-EMF data was gathered and calculated onboard the Arduino in bit values. These values was then subsequently converted to voltages in post-processing. This is shown in Figure 3.24 and forms the basis for identifying different states.

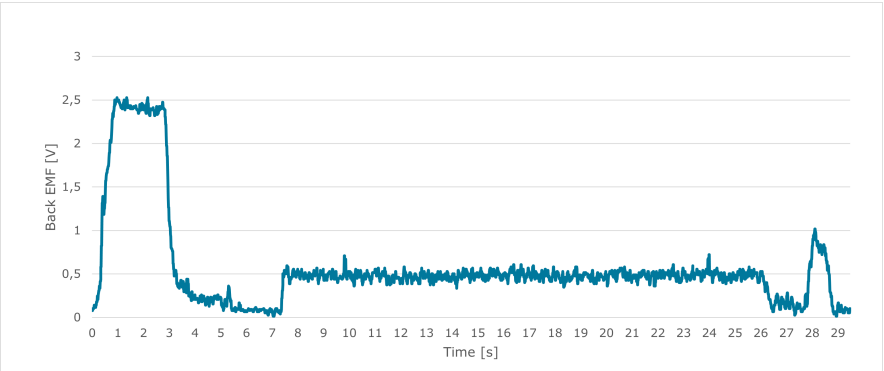

Figure 3.24 The absolute value of the back-EMF, as seen over an entire operation cycle.

The next step was to generate smooth transitions between states by adding the mapping and adjusting the echo rejection calculation rate to a fixed value. A reference graph illustrating the variable delay on the y-axis and with time on the x-axis was constructed for this purpose, as shown in Figure 3.25. The graph will provide an easy means of comparing the results before and after the smoothed out transitions.

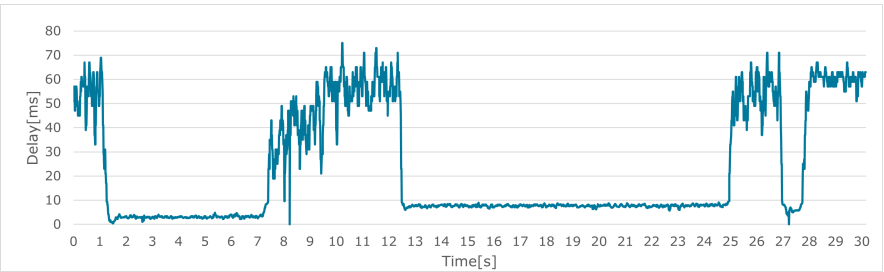

Figure 3.25 The echo rejection delay as seen before filtering.

As can be seen in the Figure 3.25, when the velocity is near zero and the door is stationary, the echo rejection time is approximately 70 ms. As the frequency reaches around 200 Hz, there is a change in state, mapping from 75 ms to 17.5 ms, down to

mapping from 3.25 ms to 2.4 ms. This is evident from the graph at 1 second, where the echo rejection time stabilizes at about 2.4 ms. It then switches back to mapping from 75 ms to 17.5 ms as the frequency approaches 0 Hz, which occur at around 7 *s*. Upon reaching 12 s, the frequency lies around 100 Hz, prompting a state change to map from 17.5 ms to 7.5 ms, resulting in a echo rejection time of approximately 7.5 *ms*.

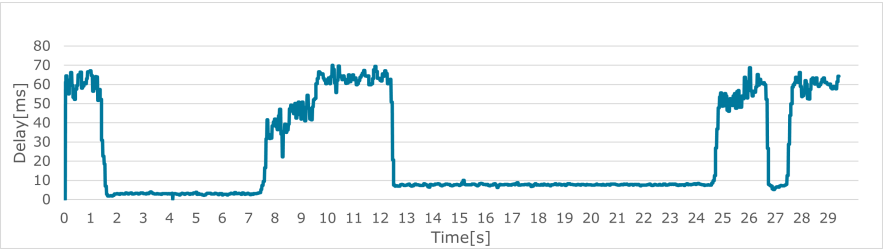

Figure 3.26 The echo rejection delay as seen after filtering.

Figure 3.26 shows the echo rejection time for the entire run with weighted values, showing a smoother signal. As shown in the figures 3.27 to 3.31, data representing runs with the new features over different door velocity settings are illustrated. The first ripple peak represents the ripple generated from the opening sequence of the door, while the second peak corresponds to the closing sequence of the door. These peaks provide a clear visualization of how the voltage ripple counting vary with changes in velocity setting and how its accuracy changes.

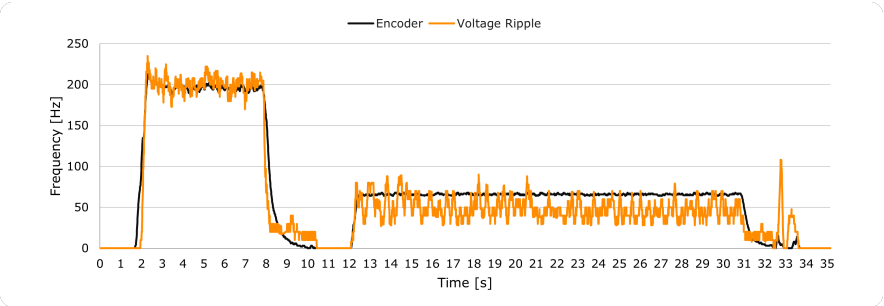

Figure 3.27 The observed ripple at 300 mm/s and 100 mm/s.

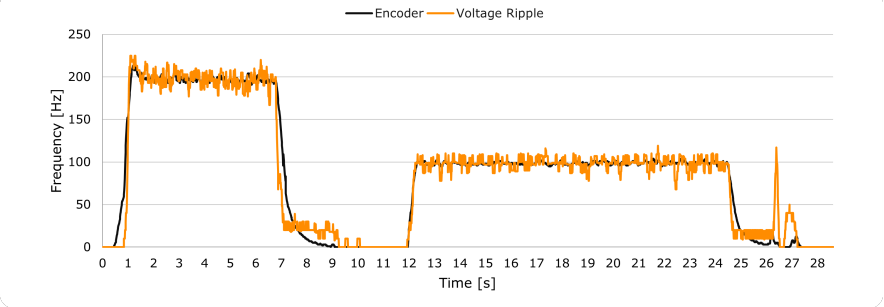

Figure 3.28 The observed ripple at 300 mm/s and 150 mm/s.

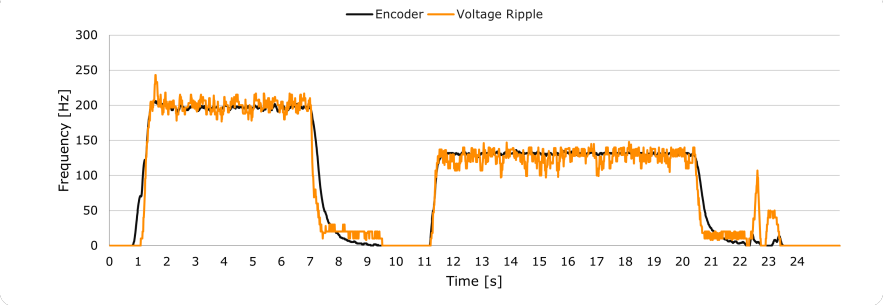

Figure 3.29 The observed ripple at 300 mm/s and 200 mm/s.

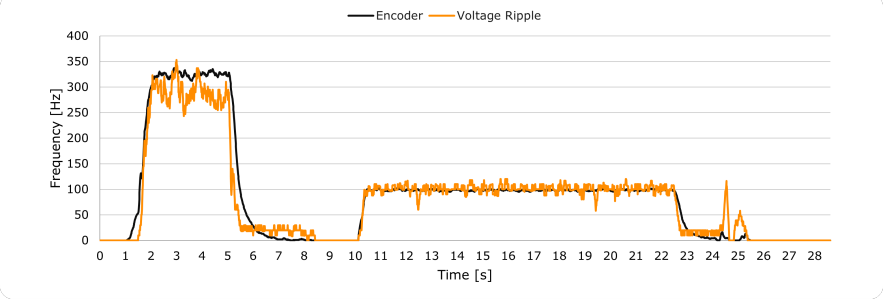

Figure 3.30 The observed ripple at 500 mm/s and 150 mm/s.

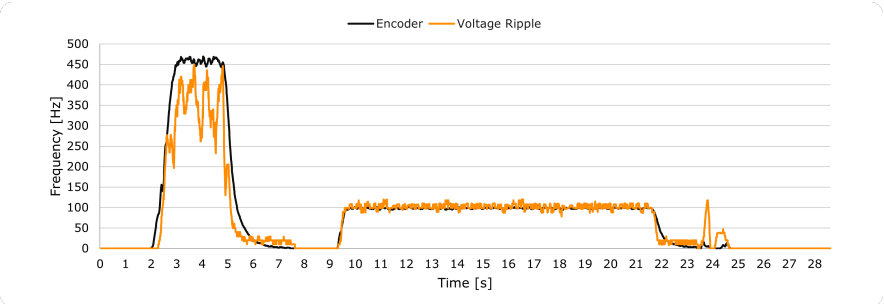

Figure 3.31 The observed ripple at 700 mm/s and 150 mm/s.

3.12 Step 12: Fine-Tuning echo rejection

Having implemented an adjustable echo rejection technique, there was still an observable difference between the velocities registered by the onboard encoder and velocity estimation given by the counted ripples. This experiment goes through the steps of fine tuning the echo rejection software in order to provide better results.

3.12.1 Step 12: Method

After examining the results at different velocities, a need for more states was identified as some velocities of the door, like 100 mm/s, differed from the encoder readings. By implementing another state handling lower velocities than 150 mm/s, the accuracy of the ripple velocity estimation was improved. However, to handle higher velocities such as 700 mm/s, another approach is needed due to the reference of 8 samples being too slow and the comparator hysteresis being too high.

3.12.2 Step 12: Result

After implementing the low velocity state, the results for velocities such as 100 mm/s improved, as can be shown in Figure 3.32, contrary to the Figure 3.27. However, the problem with higher velocities still exists, with some ripples being missed, as can be shown in Figure 3.32.

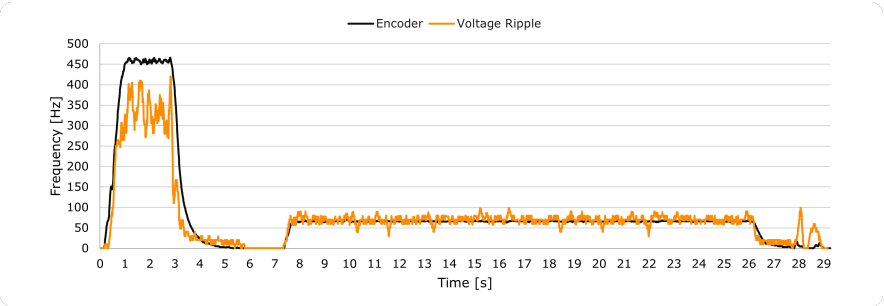

Figure 3.32 The observed ripple at 700 mm/s and 100 mm/s.

To ensure that previous results did not change with the implementation of the new state, a run with the new settings was done at 300 mm/s and 150 mm/s, as shown in Figure 3.33. As can be seen, the changes did not affect the accuracy.

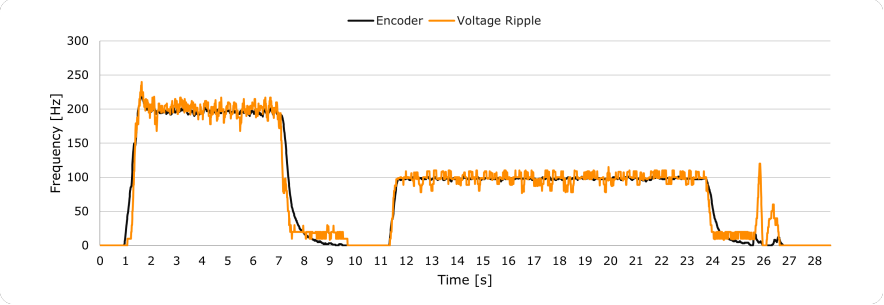

Figure 3.33 The observed ripple at 300 mm/s and 150 mm/s.

To visualize the problems encountered during the run at 700 mm/s, some focused pictures of the problem areas are shown in figures 3.34 and 3.35. As can be seen, the reference is not fast enough to accurately track some of the ripples, and in combination with the hysteresis being too large, this results in some voltage ripples not being counted.

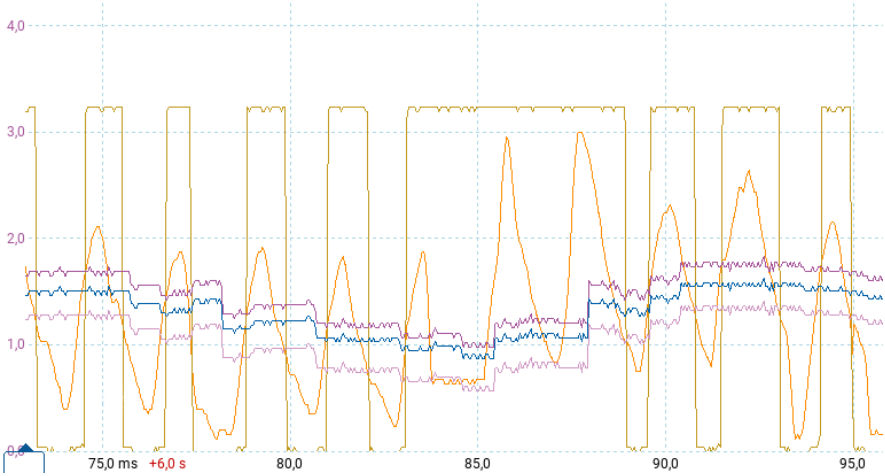

Figure 3.34 An instance where the hysteresis fails to capture the ripple.

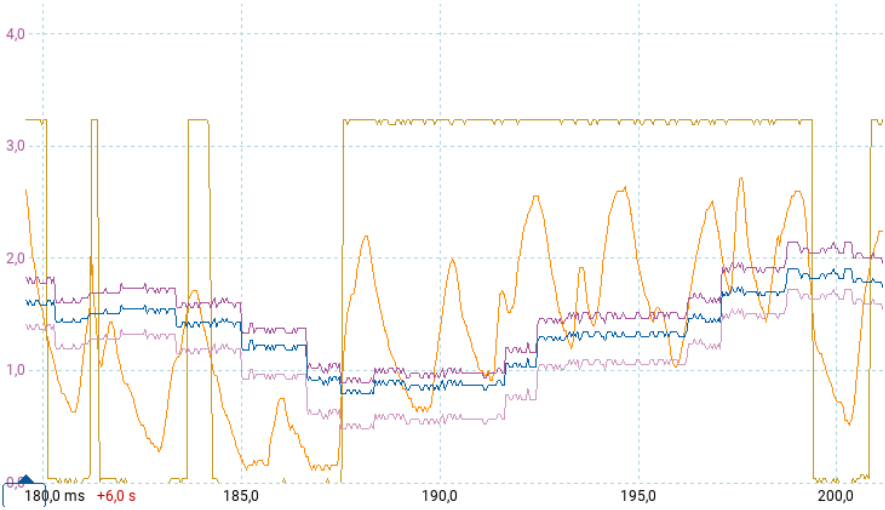

Figure 3.35 An instance visualizing how the reference is too slow to capture the ripple.

4

Results

In the following section, the results produced by the experiments described in the experimental work are compiled into a single result, summarizing the outcomes of the experiments and providing an overview of the circuit structure as well as the resulting voltage ripple.

4.1 Circuit schematics

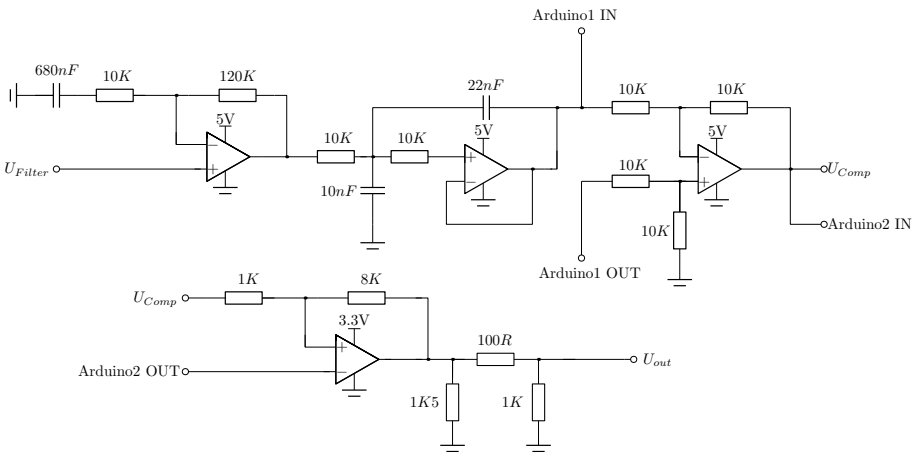

Figure 4.1 The resulting circuit used in the project.

4.2 Resulting voltage ripple counting

As discussed in chapter 3, a voltage ripple could be identified, filtered an amplified so as to be registered by a suitable microcontroller. The results, given the circuit structure displayed in Figure 4.1, is visualized in Figure 4.2. In the figure, the enlarged ripple is visualized by the orange signal, the moving reference produced by

the Arduino is displayed as the blue signal, and the output from the comparator is the yellow square wave.

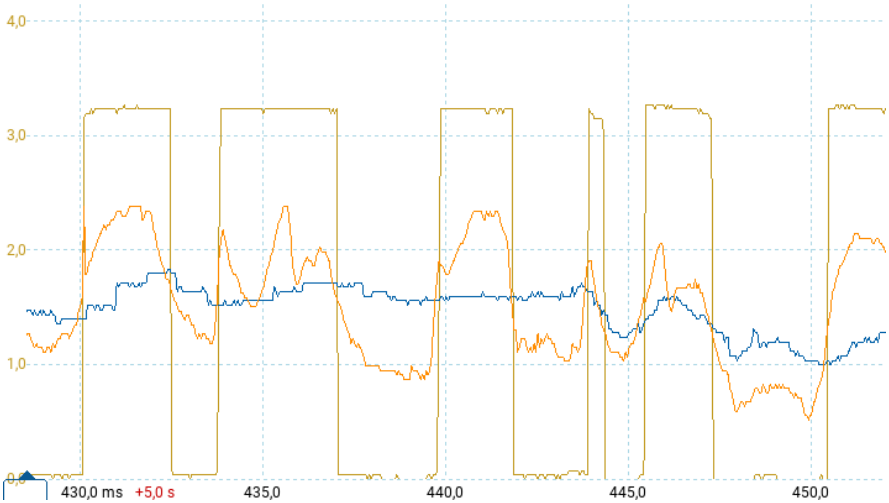

Figure 4.2 The resulting comparator output using a moving reference.

With the results shown in Figure 4.2, the ripple could be counted utilizing an Arduino interrupt routine connected to a digital pin triggering on rising flanks. By utilizing an echo rejection technique, false readings could be minimized, leading to the resulting velocity estimations displayed in Figure 4.3 where it is compared against the velocity as given by the high resolution encoder installed onboard the motor and registered in parallel aboard the Arduino.

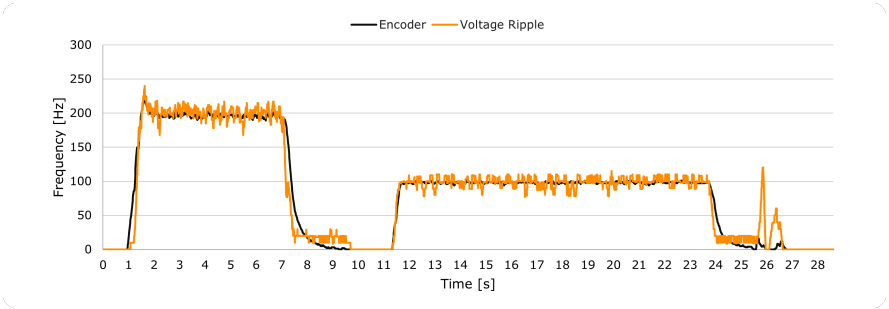

Figure 4.3 The observed ripple at 300 mm/s and 150 mm/s.

While the velocity estimation based upon the voltage ripple is reasonably close to that of the onboard encoder during door velocites of both 150 mm/s and 300

mm/s, the results start to differ when the door velocity increases. This is visualized below in Figure 4.4.

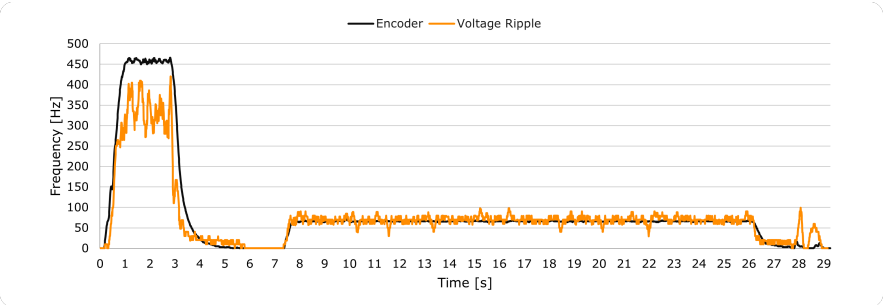

Figure 4.4 The observed ripple at 700 mm/s and 100 mm/s.

The resulting velocity estimations over the different velocities are compiled in Table 4.1 below.

Table 4.1 The observed ripple based velocity over different frequencies.

Door velocity [mm/s]	Rotational velocity (shaft) [rpm]	Expected ripple frequency [Hz]	Observed ripple frequency [Hz]
700	2300	460	275-420
500	1650	330	250-350
300	1000	200	200
200	665	133	110-135
150	500	100	100
100	330	66	60-70

4.3 Associated problems

A number of problems could be identified during the experimental work which compromised the accuracy of the resulting ripple-based velocity estimation. One of the problems was the limited sampling rate of the Arduino ADC. Another problem was related to the fact that variations in the voltage ripple signal led to some ripples being missed, and some counted additional times. Furthermore, the velocity was not properly detected during deceleration or braking phases of the motor, which was caused by other large disturbances in the signal.

The slow reading of the Arduino ADC produced by the onboard AnalogRead() function resulted in loss of accuracy at higher door velocities. The effects of this are

visualized in Figure 4.5 where the door travels at 700 mm/s, resulting in a ripple at 460 Hz. The maximum ADC reading frequency that could be achieved utilizing the standard function with an accuracy of 12 bits was 1167 Hz. This can also be seen in Figure 4.5, as the reference wave value is updated at the same frequency. In the figure, the reference value was produced as a moving average over 8 samples. This means that whilst the reference could be made to move in greater steps than visualized by utilizing a smaller sample buffer, it would have no effect on the underlying issue of the slow reading frequency for any bit resolution previously discussed.

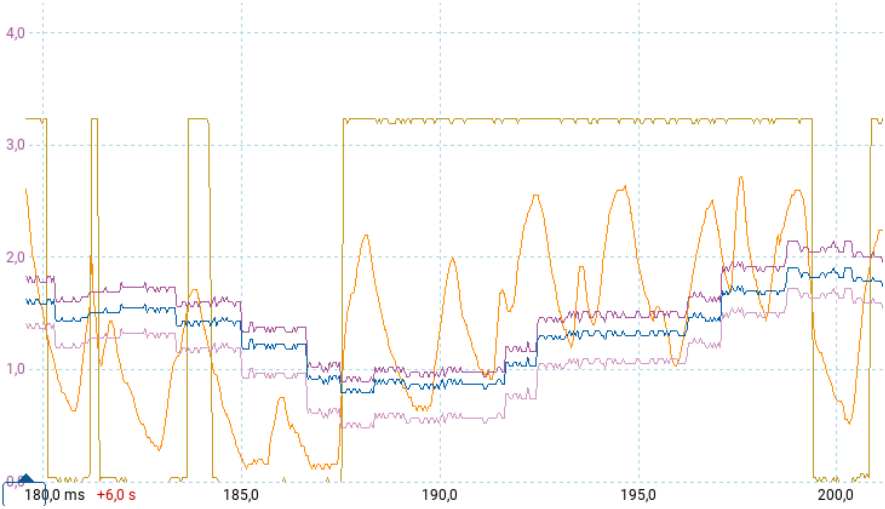

Figure 4.5 An instance visualizing how the reference is too slow to capture the ripple.

As previously mentioned, the sampling rate at 12 bit resolution was 1167 Hz. Given the observed ripple frequency of 460 Hz, a resulting ripple resolution of 2.54 samples per ripple cycle is yielded.

Regarding the fact that some ripples were missed, whilst others were counted several times, the implementation of the echo rejection technique utilizing the back EMF as guard condition for state transition only partly solved the problem. While the problem was reduced, the result was still not perfect. As discussed in section 3.10, the method of using a delay to reject additional readings ran the risk of turning the program in to a self fulfilling prophecy. This is exemplified in Figure 4.6.

Figure 4.6 An instance visualizing how some inputs are rejected, and associated problems.

As can be seen in Figure 4.6, the orange line is the ripple, the blue line is the comparator reference, the yellow line is the comparator output, and the green line is the comparator outputs acknowledged by the echo rejection method, as visualized with a digital pin aboard the Arduino. Looking at the first 6 ripples starting from 14.86 s, it is clear that additional spikes are rejected, and that the counting works as intended. However, at around 14.915 s a ripple is registered earlier than usual, leading to a lingering spike at 14.924 s being registered as an independant ripple. Because of this the subsequent ripple is rejected at 14.931 s and the echo rejection only lines up with the ripples again at the ripple after the rejected one, at 14.938 s. In this case, the number of counted ripples is correct, although the underlying reason is a mistaken ripple.

During the experimental work, a pattern could be detected during braking, or desecration, cycles. This is exemplified in Figure 4.3. In the figure, the ripples are counted as intended during acceleration and in the constant velocity zones. At both 7.5 s and 24 s the number of counted ripples sharply drops, however, and in both cases results is an estimated velocity of roughly 20 Hz. The reason behind this can be seen in Figure 4.7, where another unknown, uniform and alternating ripple pattern is observed. These are likely not waves caused by the commutation process in the motor, and can therefore not be used to estimate the velocity. During the deceleration phase these waves are, however, the only wave structures that are easily observed. Because of this, they are picked up by the comparator and subsequently the Arduino.

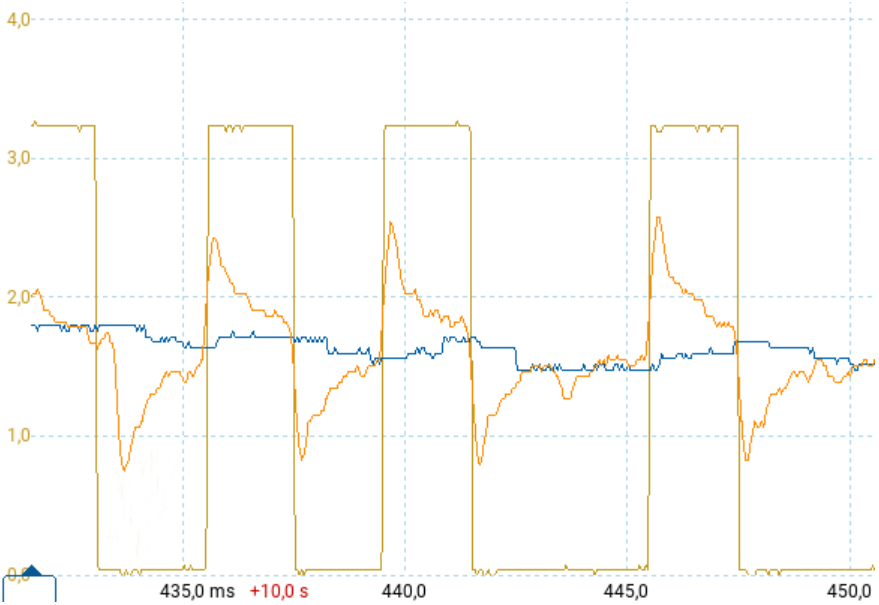

Figure 4.7 Visualized is the low frequency pattern that emerged during deceleration.

5

Discussion

The final results and iterations differed slightly from the expectations outlined in the problem description (section 1.2), which also aimed to control the door by replacing the encoder signal. Nevertheless, removing the encoder and achieving accurate results proved to be sufficiently challenging. The solution of tracking the commutation frequency first in the form of current ripples (Figure 3.2), and later in the form of voltage ripples (Figure 3.4), showed predominantly positive results at lower velocities, where the accuracy met our goal. Expanding the solution to include an actual sliding door system also produced positive results at low to medium velocities up to 250 Hz, or 1250 rpm. At higher velocities, above 1250 rpm, the results were not as good due to a number of unresolved problems. This indicates the need for further optimization to handle faster operational speeds effectively. One such improvement would be to increase the ADC sampling speed, as this was the biggest limiting factor identified.

Compared to the method of using nonlinear back-EMF integration for position control, counting voltage ripples turned out be a viable alternative if it can be done accurately. The fact that voltage or current ripples are the product of a periodic commutation phenomenon provides a means to counteract the drift in the position estimation that is prone to occur when integrating over the velocity. This is a problem especially when utilizing back-EMF techniques as mentioned in section 2.3. This means that, with further tuning, voltage ripple counting might serve as a replacement or complement to existing encoder solutions depending on the implementation. With this said, there are both advantages and disadvantages with the solution presented in this thesis.

The advantages of the implemented circuitry includes its cost-effectiveness compared to high-resolution encoders that produce hundreds of pulses per rotation. This does, however, rely upon the fact that an adequate microcontroller, or equivalent hardware, that can support the software and is available for additional tasks already exists.

The implemented system can also process signals quickly with, given appropriate tuning, minimal delay and focuses solely on the desired ripple. It does this without altering it, thus providing an accurate reading of the ripple for further use. On the other hand, the disadvantages includes the dependency on a relatively fast microcontroller with an accurate ADC that can handle various signals. The initial high-pass filter adds complexity to the circuit, for which simpler solutions may exist to lower the voltage range. Additionally, the comparator was designed as a non-inverting comparator, which produces hysteresis with a shifting saturation value. Using a constant saturation may provide better results and can be achieved by switching input terminals for the reference and ripple signal. Another solution would be to simple utilize Equation 3.8 in software so as to always produce a reference with an appropriate hysteresis that centers around the ripple. The largest disadvantage at the moment, however, is the way the measurements are made from the H-bridge, where a differential reading across both legs holds the promise of enabling accurate measurements to be made during braking phases as well. This is something that is not possible at the moment and that can not be addressed by means of software. Addressing these disadvantages of the circuitry and dealing with the existing software problems should enable further improvements to be achieved. Such software problems include the analog reading speed, which should be possible to increase by writing new analogue read functions tailored to the problem. Moreover, large improvements can likely be achieved by removing the back-EMF as a safeguard for state changes and further investigating the problem with the self-fulfilling prophecy.

5.1 Future work

For future development, a more sophisticated software environment is needed. This should include merging the two microcontrollers into one unit and utilizing the multiple unused clocks in the SAMD21 microcontroller to operate various functions concurrently. Additional circuitry for the Digital-to-Analog Converter (DAC) should be designed to leverage the numerous PWM pins on the SAMD21.

For real-time application use, an approach to determine the mean current voltage ripple frequency should be implemented. This would involve calculating the ripple frequency based on the counted voltage ripples and elapsed time, followed by filtering the result with a generic rolling mean function. This analysis should be integrated directly into the function that outputs the ripple frequencies, rather than being performed post hoc. It is also evident that optimizing the ADC setup or using alternative methods to increase sampling efficiency is critical. This was seen due to the given constraints of the current system only being able to sample at 1167 Hz with a 12 bit resolution. A sampling rate that turned out to be insufficient for

capturing the high-frequency voltage ripples accurately. This can be exemplified with the ripples that were observed at 460 Hz where the system, at the moment, is able to sample a maximum of three times per ripple which is obviously to low. In general, enhancing the ADC's capability to handle higher frequencies without losing resolution could significantly improve the accuracy of ripple detection and velocity estimation. Enhancing ADC performance on the SAMD21 can be done by adjusting its settings to move away from standard Arduino functions. This includes enabling Direct Memory Access (DMA) for efficient data transfer, using overflow features to manage high-frequency data, and increasing the ADC clock speed to boost sampling rates. These changes are just a few options how to improve it and can significantly improve the system's speed and accuracy in capturing rapid voltage ripple changes.

In the hardware section, the critical error of only utilizing a single node on the H-bridge, point "a" in Figure 2.7, needs to be addressed for more robust readings, as well as possibly remedying and correcting the unknown pattern that appears during braking as shown in Figure 4.7. Not doing so means the ripple counting technique is unusable during a large portion of the operational range of the door, due to the braking signals being erratic with no trace of commutation ripples to be observed.

It is also important to find another solution for lowering the voltage besides using a high-pass filter and fixing the shifting saturation level to be constant. Additionally, conducting a cost analysis of the circuitry and testing the setup and code on a ASSA ABLOY control unit is crucial to determine if it can be implemented on their current system, or if further hardware changes needs to be made to enable the technique to be incorporated in to their sliding door systems in the future. This would ensure that the solution is not only technically viable but also economically feasible.

Unfortunately, due to the time limitations of the project, the scenarios and speed of voltage ripple counting were restricted to lower and mid-range speeds in undisturbed scenarios. This meant there was no human interaction except for initiating the door sequence, which differs from potential commercial applications. Furthermore, the available time was insufficient to fully explore and develop the technique to completely replace the encoder.

6

Conclusion

To conclude the results of the report and this project, and in turn answer the questions described in the problem description (section 1.2). To implement an adequate control system without using an encoder, a similar measurement system to an encoder needs to be created without actually using one. This can be achieved by utilizing the phenomenon of voltage ripples found in subsection 3.2.1, which can be utilized in a manner similar to that of an encoder. Accuracy in measurements is achieved by filtering the signal to remove unwanted noise where after the ripple component is amplified. For the brushed DC motor, the implementation involves the use of filter circuits, amplifiers, comparators, differential amplifiers, and microcontrollers that samples a signal originating from the H-bridge used motor control aboard the ASSA ABLOY control unit. However, the system's performance has its limitations such as the current circuit configuration and the ADC's sampling capabilities, particularly during high-speed operations, deceleration and braking. Leading to accurate measurements only being feasible at medium to low velocities and given that operation is conducted within an ideal environment.

To further enhance the accuracy and reliability of the system, several adjustments are necessary. The improvements should focus on refining the circuit schematics and components to better manage noise and improve signal clarity, while at the same time simplifying the circuit as much as possible but still maintaining its reading accuracy. It is also crucial to expand the measurements across the H-bridge to enable readings to be made during all phases of operation, including opening, closing, standstill and braking phases. In contrast to the current implementation that does not work during deceleration or braking phases. On the software side, it is important to enhance the ADC capabilities to increase sampling rates and to implement a more sophisticated signal processing to identify all motor ripples in the noise. This is especially important during high-velocity operations.

With the current solution it is not possible to implement an adequate control system for motor control using solely voltage ripples as a replacement for an encoder.

But this thesis has proved that further research based upon the findings produced in this report might enable this to be done in the future.

Bibliography

[1] G. Rizzoni and J. Kearns. *Principles and Applications of Electrical Engineering*. International edition, 6th edition. McGraw-Hill Education, 2016. ISBN: 978981457741.

[2] G. R. Slemon. *Direct-current commutator motors*. Fact-checked and reviewed by Britannica editors. 2023. URL: https://www.britannica.com/technology/electric-motor (visited on 2024-02-08).

[3] M. Alaküla and P. Karlsson. *Power Electronics: Devices, Converter, Control and Applications*. Department of Industrial Electrical Engineering and Automation, 2019.

[4] A. Automation. *Encoders guide*. 2021. URL: https://www.anaheimautomation.com/manuals/forms/encoder-guide.php (visited on 2024-02-15).

[5] M. Vidlak, L. Gorel, P. Makys, and M. Stano. "Sensorless speed control of brushed dc motor based at new current ripple component signal processing". *Energies* **14** (2021), p. 5359.

[6] P. Radcliffe and D. Kumar. "Sensorless speed measurement for brushed dc motors". *IET Power Electronics* **8**:11 (2015), pp. 2223–2228.

[7] P. Microdrives. *Measuring rpm from back emf*. 2021. URL: https://www.precisionmicrodrives.com/ab-021 (visited on 2024-02-08).

[8] A. Abacan, M. L. Canada, M. Gomez, and M. T. Inc. *Sensorless position control of brushed dc motor using ripple counting technique*. 2019. URL: https://ww1.microchip.com/downloads/en/AppNotes/Sensorless-Position-Control-of-Brushed-DC-Motor-Using-Ripple-Counting-Technique-00003049A.pdf (visited on 2024-05-29).

[9] R. Mancini. *Op amps for everyone design guide*. 2002. URL: https://web.mit.edu/6.101/www/reference/op_amps_everyone.pdf (visited on 2024-04-26).

[10] A. Ahmad. *Calculating rc low-pass filter cut-off frequency and transfer function*. 2023. URL: https://eepower.com/technical-articles/calculating-rc-low-pass-filter-cut-off-frequency-and-transfer-function/ (visited on 2024-04-26).

[11] E. Tutorials. *Passive low pass filter*. URL: https://www.electronics-tutorials.ws/filter/filter_2.html (visited on 2024-04-26).

[12] E. Tutorials. *Sallen and key filter*. URL: https://www.electronics-tutorials.ws/filter/sallen-key-filter.html (visited on 2024-04-26).

[13] T. Instruments. *Single-supply, 2nd-order, sallen-key low-pass filter circuit*. 2021. URL: https://www.ti.com/lit/an/sboa226/sboa226.pdf (visited on 2024-04-26).

[14] Renesas. *Op-amps, comparator circuit*. 2024. URL: https://www.renesas.com/us/en/support/engineer-school/electronic-circuits-03-op-amps-comparator-circuit (visited on 2024-04-26).

[15] A. Kay and T. Claycomb. *Comparator with hysteresis reference design*. 2013. URL: https://www.ti.com/lit/ug/tidu020a/tidu020a.pdf (visited on 2024-04-26).

[16] O. Samuelsson. *Op-förstärkare*. Lecture notes presented in the basic course in electrical engineering. Course code: EEIF35. 2021.

[17] A. Tantos. *H-bridges – the basics*. 2011. URL: https://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/ (visited on 2024-03-16).

[18] Arduino.cc. *Mkr zero dokumentation*. 2024. URL: https://docs.arduino.cc/hardware/mkr-zero/ (visited on 2024-04-24).

[19] M. Technology. *Sam d21/da1 family datasheet*. 2021. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/SAM-D21DA1-Family-Data-Sheet-DS40001882G.pdf (visited on 2024-04-24).

[20] M. Facchin and Arduino. *Wiring_analog.c*. 2020. URL: https://github.com/arduino/ArduinoCore-samd/blob/master/cores/arduino/wiring_analog.c (visited on 2024-04-24).

[21] Arduino.cc. *Digital pins*. 2023. URL: https://docs.arduino.cc/learn/microcontrollers/digital-pins/ (visited on 2024-04-25).

[22] Arduino.cc. *High | low*. Revision: 2024/02/15. 2024. URL: https://www.arduino.cc/reference/en/language/variables/constants/highlow/ (visited on 2024-04-25).

[23] N. Gammon. *Interrupts*. 2016. URL: https://www.gammon.com.au/interrupts (visited on 2024-04-25).

[24] S. W. Smith. *The scientist and engineer's guide to digital signal processing*. Copyright © 1997-2011 by California Technical Publishing. 1997. URL: https://www.dspguide.com/pdfbook.html (visited on 2024-05-05).

[25] GeeksforGeeks. *How to calculate the weighted mean?* Last updated: April 13, 2022. 2022. URL: https://www.geeksforgeeks.org/how-to-calculate-the-weighted-mean/ (visited on 2024-05-05).

[26] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Second Edition. MIT Press, 2017. ISBN: 978-0-262-53381-2.

7

Appendix A: Arduino Code

Arduino C++ Code for Differential Amplifier Signal Processing

```
1
2  #include <SPI.h>
3  #include <stdint.h>
4  #include <Arduino.h>
5
6  //*** Pins ***//
7  #define PIN_ARDUINO_DAC_OUT A0 // The pin that produces the
       DAC signal that is fed to the differential amplifier.
8  #define PIN_ARDUINO_DAC_IN A1  // The source signal pin that
        is to be filtered and reproduced as the DAC output.
9
10 //*** Constants ***//
11 #define BIT_RESOLUTION 12                    // The analog
       reading bit resolution.
12 #define BIT_RESOLUTION_TO_INT 4096           // Conversion
       factor from bit resolution to integer.
13 #define OFFSET_DIFF_AMPLIFIER 1240           // Increase
       the duty-cycle of the DAC to keep the entire
       measurement signal above zero. Necessary as the
       Amplifier is powered on a single rail.
14 #define SAMPLE_NBR_DAC_BIT 4                 // Bit size for
       calculating the size of the sample array for rolling
       mean.
15
16 //*** Rolling Mean Variables for Differential Amplifier
       Output ***//
17 volatile int UMotor2DiffAmpValue_RollingMean = 0; // Rolling
        mean of the differential amplifier output voltage.
18 int UMotor2DiffAmp_TotalVectorValue = 0;          // Total
       sum of values for calculating the rolling mean.
```

```
19 int UMotor2DiffAmp_Index = 0;                    // Current
       index in the rolling mean array.
20 int UMotor2DiffAmp_Mean[1 << SAMPLE_NBR_DAC_BIT]; // Array
       to hold mean values for rolling calculations.
21
22 // %%%%%%%%%%%%%%%%%%%%%%%% SETUP %%%%%%%%%%%%%%%%%%%%%%%%
23 void setup()
24 {
25   // Initialize and set analog resolution to BIT_RESOLUTION,
         12 bits (Standard value).
26   analogReadResolution(BIT_RESOLUTION);
27   analogWriteResolution(BIT_RESOLUTION);
28
29 }
30
31 // %%%%%%%%%%%%%%%%%%%%%%%% LOOP %%%%%%%%%%%%%%%%%%%%%%%%
32 void loop()
33 {
34 UMotor2DiffAmp_RollingMean();
35 offset2DiffAmplifier_DAC();
36 }
37
38 // This function produces the filtered value of the motor
       voltage, as seen after the first stage ampliier,
39 // via the DAC. This is subsequently fed to the positive
       input off the differential amplifier,
40 // where it is subtracted from the hardware filtered source
       signal.
41 // This will leave the sought after ripple component as the
       only output by removing the DC offset.
42 void offset2DiffAmplifier_DAC()
43 {
44   int output_DAC = UMotor2DiffAmpValue_RollingMean;
45   //Adding an offset(1240) to increase the reference value
       above zero and making sure it does not surpass
       BIT_RESOLUTION_TO_INT(4096)
46   int outputMappedAdjusted_DAC = max(0,min(output_DAC +
       OFFSET_DIFF_AMPLIFIER, BIT_RESOLUTION_TO_INT));
47   analogWrite(PIN_ARDUINO_DAC_OUT, outputMappedAdjusted_DAC)
       ;
48 }
49
50 // This function provides software filtering of the measured
       motor voltage as it appears after
51 // the first OP stage. Is used to produce the DAC signal.
52 void UMotor2DiffAmp_RollingMean()
```

```
{
  // Reduce the total value of the stored vector with the
    oldest value.
  UMotor2DiffAmp_TotalVectorValue -= UMotor2DiffAmp_Mean[
    UMotor2DiffAmp_Index];
  // Replace the oldest value with newly read value.
  UMotor2DiffAmp_Mean[UMotor2DiffAmp_Index] = analogRead(
    PIN_ARDUINO_DAC_IN);
  // Add the new value to the total value of the stored
    vector
  UMotor2DiffAmp_TotalVectorValue += UMotor2DiffAmp_Mean[
    UMotor2DiffAmp_Index];
  // Increment the index with wrap-around
  UMotor2DiffAmp_Index = (UMotor2DiffAmp_Index + 1) % (1 <<
    SAMPLE_NBR_DAC_BIT);
  // Divide the total with the sample size
  UMotor2DiffAmpValue_RollingMean = (
    UMotor2DiffAmp_TotalVectorValue) >> SAMPLE_NBR_DAC_BIT;
}
```

Listing 7.1 Arduino class DCdiffHandler used in the differential amplifier

Arduino C++ Code for the compartor and voltage ripple counting

```

#include <SPI.h>
#include <stdint.h>
#include <Arduino.h>

//*** Constants ***//
#define BIT_RESOLUTION 12  // The analog reading bit
    resolution.

//*** Pin Definitions ***//
#define PIN_ENCODER_IN 0             // The encoder input
    pin. Later set to detect change via ISR.
#define PIN_COMPARATOR_IN 1          // The comparator
    output pin.
#define PIN_ARDUINO_DAC_OUT A0       // The DAC output pin
    connected to differential amplifier.
#define PIN_ARDUINO_DAC_IN A1        // The source signal
    pin for DAC output.
#define PIN_DIFFAMP_OUTPUT A2        // The output pin from
    the differential amplifier.
```

```
15 #define PIN_U_MOTOR_1_MCU A5         // Motor control pin 1.
16 #define PIN_U_MOTOR_2_MCU A3         // Motor control pin 2.
17 #define PIN_I_MOTOR A4               // Motor current
       measurement pin.
18 #define PIN_COMPARATOR_OUTPUT 6      // The comparator
       output pin.
19
20 //*** Encoder Variables ***//
21 int encoderPinValue = 0;
22 int encoderLastPintValue = 0;
23 int encoderCount = 0;
24 int encoderSwitchCount = 0;
25 unsigned long encoderCurrentTime = 0;
26 unsigned long encoderLastTime = 0;
27
28 //*** Comparator Variables ***//
29 int comparatorPinValue = 0;
30 int comparatorLastPinValue = 0;
31 int comparatorCount = 0;
32 int lastComparatorCount = 0;
33 unsigned long lastComparatorTime = 0;
34
35 //*** Motor Control and Measurement Variables ***//
36 int StartGatheringDataFlag = 0;
37 int U_MOTOR_1_MCU = 0;
38 int U_MOTOR_2_MCU = 0;
39 int I_MOTOR = 0;
40
41 //*** Rolling Mean Calculation for Back EMF and Filter
       Output ***//
42 #define SAMPLE_NBR_BACKEMF_BIT 2
43 #define SAMPLE_NBR_BACKEMF (1 << SAMPLE_NBR_BACKEMF_BIT)
44 volatile int UMotorBackEMFValue_RollingMean = 0;
45 volatile int UMotorBackEMFValue_LastReadValue = 0;
46 int UMotorBackEMF_TotalVectorValue = 0;
47 int UMotorBackEMF_Index = 0;
48 int UMotorBackEMF_Mean[1 << (SAMPLE_NBR_BACKEMF_BIT)];
49
50 volatile int UMotorFilterOutputValue_RollingMean = 0;
51 volatile int UMotorFilterOutput_LastReadValue = 0;
52
53 #define SAMPLE_NBR_2_OUTPUT_BIT 1
54 int UMotorFilterOutput_2Samples_TotalVectorValue = 0;
55 int UMotorFilterOutput_2Samples_Index = 0;
56 int UMotorFilterOutput_2Samples_Mean[1 << (
       SAMPLE_NBR_2_OUTPUT_BIT)];
```

```
57
58 #define SAMPLE_NBR_4_OUTPUT_BIT 2
59 int UMotorFilterOutput_4Samples_TotalVectorValue = 0;
60 int UMotorFilterOutput_4Samples_Index = 0;
61 int UMotorFilterOutput_4Samples_Mean[1 << (
       SAMPLE_NBR_4_OUTPUT_BIT)];
62
63 #define SAMPLE_NBR_8_OUTPUT_BIT 3
64 int UMotorFilterOutput_8Samples_TotalVectorValue = 0;
65 int UMotorFilterOutput_8Samples_Index = 0;
66 int UMotorFilterOutput_8Samples_Mean[1 << (
       SAMPLE_NBR_8_OUTPUT_BIT)];
67
68 #define SAMPLE_NBR_16_OUTPUT_BIT 4
69 int UMotorFilterOutput_16Samples_TotalVectorValue = 0;
70 int UMotorFilterOutput_16Samples_Index = 0;
71 int UMotorFilterOutput_16Samples_Mean[1 << (
       SAMPLE_NBR_16_OUTPUT_BIT)];
72
73 //*** Timing and Debounce Variables ***//
74 unsigned long serialOutPutTime = 0;
75 unsigned long backEMFTime = 0;
76 unsigned long comparatorCurrentTime = 0;
77 volatile int echoRejectionDelay = 0;
78 int echoRejection_LastReadValue = 0;
79 int echoRejection_CurrentReadValue = 0;
80 unsigned long echoRejectionTime = 0;
81
82 //*** Weights for Rolling Means ***//
83 double weightLast = 0.6;
84 double weightNew = 0.4;
85
86 //*** States based on Back EMF ***//
87 enum BackEMFState {
88   VERY_HIGH_BACK_EMF,
89   HIGH_BACK_EMF,
90   MEDIUM_BACK_EMF,
91   LOW_BACK_EMF,
92   VERY_LOW_BACK_EMF
93 };
94
95
96
97 // %%%%%%%%%%%%%%%%%%%%%%%% SETUP %%%%%%%%%%%%%%%%%%%%%%%%
98 void setup()
99 {
```

```
100
101   // Encoder interrupt counter pin [Pin 0].
102   pinMode(PIN_ENCODER_IN, INPUT);
103   attachInterrupt(digitalPinToInterrupt(PIN_ENCODER_IN),
        ISR_encoder, RISING);
104
105   // The counted Ripples visualized as a square wave output.
         Only for oscilloscope use. [pin 6].
106   pinMode(PIN_COMPARATOR_IN, INPUT);
107   attachInterrupt(digitalPinToInterrupt(PIN_COMPARATOR_IN),
        ISR_comparator, RISING);
108
109   // Initialize and set analog resolution to BIT_RESOLUTION,
         10 bits (Standard value).
110   analogReadResolution(BIT_RESOLUTION);
111
112
113   // Initializes serial communication at baud rate 115200.
114   Serial.begin(9600);
115   serialOutPutTime = millis();
116   backEMFTime = millis();
117   lastComparatorTime = micros();
118 }
119
120 // %%%%%%%%%%%%%%%%%%%%%%%% LOOP %%%%%%%%%%%%%%%%%%%%%%%%
121 void loop()
122 {
123   UMotorBackEMF_RollingMean();
124   adjustableEchoRejectionStateMachine();
125   UMotorFilterOutput_RollingMean();
126   comparatorOutPrint();
127 }
128
129
130 // Interrupt Service Routine (ISR) for the comparator.
131 // Triggered on rising edges detected on the comparator's
        input pin.
132 void ISR_comparator()
133 {
134     // Record the current time in microseconds at the moment
         of interrupt.
135     comparatorCurrentTime = micros();
136
137     // Calculate the time difference between the current and
         the last interrupt.
138     unsigned long timeDifference = comparatorCurrentTime -
```

```
    lastComparatorTime;
139
140     // Check if the current interrupt occurred within the
        echo rejection delay window.
141     if (timeDifference <= echoRejectionDelay)
142     {
143       return;// Exit the ISR if the time since the last
        interrupt is too short.
144     }
145
146     // Increment the comparator count to track how many
        times the comparator has triggered.
147     comparatorCount++;
148
149     // Update the last comparator time to the current time
        after a valid event.
150     lastComparatorTime = comparatorCurrentTime;
151 }
152
153 // Interrupt Service Routine (ISR) for the encoder.
154 // Triggered on rising edges detected on the encoder's input
       pin.
155 void ISR_encoder()
156 {
157     // Record the current time in microseconds at the moment
       of interrupt.
158     encoderCurrentTime = micros();
159
160     // Calculate the time difference between the current and
       the last interrupt.
161     unsigned long encoderDeltaTime = encoderCurrentTime -
        encoderLastTime;
162
163      // Check if the current interrupt occurred within the
        debounce delay window.
164     if (encoderDeltaTime < 10)
165     {
166       return; // Exit the ISR if the time since the last
        interrupt is too short.
167     }
168
169     // Increment the encoder count to keep track of movement
       or rotation.
170     encoderCount++;
171
172      // Update the last comparator time to the current time
```

```
    after a valid event.
    encoderLastTime = encoderCurrentTime;
}


void comparatorOutPrint()
{
  // Calculate the time difference since the last serial output
  unsigned long timeDifference = millis() - serialOutPutTime;

  // Check if enough time has passed since the last output (minimum interval: 10 milliseconds)
  if (timeDifference >= 10)
  {
    // Disable interrupts to ensure serial printing is not interrupted
    noInterrupts();

    // Print various data separated by commas
    Serial.print(timeDifference);
    Serial.print(",");
    Serial.print(comparatorCount);
    Serial.print(",");
    Serial.print(encoderCount);
    Serial.print(",");
    Serial.print(UMotorBackEMFValue_RollingMean);
    Serial.print(",");
    Serial.print(echoRejectionDelay);
    Serial.println("");

    // Enable interrupts after printing is done
    interrupts();

    // Update the timestamp of the last serial output
    serialOutPutTime = millis();

    // Store the current comparator count for comparison with the next reading
    lastComparatorCount = comparatorCount;
  }
}

/* This function implements a generic rolling average calculation.
```

```
212  *   - ‘mean‘ is an array that stores previously read values
         .
213  *   - ‘readValue‘ is the most recent value to be included
         in the average.
214  *   - ‘index‘ specifies the current position within the ‘
         mean‘ array.
215  *   - ‘totalVectorValue‘ represents the cumulative sum of
         the values in the ‘mean‘ array.
216  *   - ‘sampleNbrBit‘ defines the number of samples,
         affecting the size of the ‘mean‘ array and the scaling
         of the average.
217  */
218 int rollingMeanGeneric(int *mean, volatile int *readValue,
         int *index, int *totalVectorValue, int sampleNbrBit)
219 {
220   // Reduce the total value of the stored vector with the
        oldest value.
221   *totalVectorValue -= mean[*index];
222   // Replace the oldest value with newly read value.
223   mean[*index] = *readValue;
224   // Add the new value to the total value of the stored
        vector
225   *totalVectorValue += mean[*index];
226   // Increment the index with wrap-around:
227   *index = (*index + 1) % (1 << sampleNbrBit);
228   // Divide the total with the sample size
229   int RollingMean = (*totalVectorValue) >> sampleNbrBit;
230
231   return RollingMean;
232 }
233
234 // Computes the rolling mean of the back electromotive force
        (EMF) readings from the motor.
235 void UMotorBackEMF_RollingMean()
236 {
237   // Calculate the time difference since the last back EMF
        reading
238   unsigned long timeDiffrence = millis() - backEMFTime;
239
240   // Check if enough time has passed since the last reading
        (minimum interval: 15 milliseconds)
241   if (timeDiffrence >= 15)
242   {
243     // Read motor voltage and current values
244     U_MOTOR_1_MCU = analogRead(PIN_U_MOTOR_1_MCU);
245     U_MOTOR_2_MCU = analogRead(PIN_U_MOTOR_2_MCU);
```

```
246     I_MOTOR = analogRead(PIN_I_MOTOR);
247
248     // Compute the back EMF value using motor voltage and
        current readings
249     UMotorBackEMFValue_LastReadValue = abs((U_MOTOR_2_MCU -
        U_MOTOR_1_MCU) - (I_MOTOR / 3) + 33); // (33 for
        reference value taken from the graph)
250
251     // Calculate the rolling mean of back EMF values
252     UMotorBackEMFValue_RollingMean = rollingMeanGeneric(
        UMotorBackEMF_Mean, &UMotorBackEMFValue_LastReadValue,
        &UMotorBackEMF_Index, &UMotorBackEMF_TotalVectorValue,
        SAMPLE_NBR_BACKEMF_BIT);
253
254     // Update the timestamp of the last back EMF reading
255     backEMFTime = millis();
256   }
257 }
258
259 // Computes the rolling mean of the motor voltage readings
        after the output of the filter and differential
        amplifier.
260 void UMotorFilterOutput_RollingMean()
261 {
262   // Read the latest voltage value from the differential
        amplifier output
263   UMotorFilterOutput_LastReadValue = analogRead(
        PIN_DIFFAMP_OUTPUT);
264
265   // Calculate the rolling mean using a generic function
266   UMotorFilterOutputValue_RollingMean = rollingMeanGeneric(
        UMotorFilterOutput_8Samples_Mean, &
        UMotorFilterOutput_LastReadValue, &
        UMotorFilterOutput_8Samples_Index, &
        UMotorFilterOutput_8Samples_TotalVectorValue,
        SAMPLE_NBR_8_OUTPUT_BIT);
267
268   // Write the calculated rolling mean to the DAC output pin
269   //
270   analogWrite(PIN_ARDUINO_DAC_OUT,
        UMotorFilterOutputValue_RollingMean);
271 }
272
273 // Function to calculate the weighted average of two values
274 int weightedAverage(int lastValue, int newValue, double
        weightLast, double weightNew)
```

```
{
  // Calculate the weighted average
  int result = (lastValue * weightLast + newValue *
    weightNew) / (weightLast + weightNew);
  return result;
}


// Function to determine the current state based upon Back
    EMF
int getBackEMFState(int backEMFValue)
{
  if (backEMFValue > 2090)
    return VERY_HIGH_BACK_EMF;
  else if ((2090 >= backEMFValue) && (backEMFValue > 1285))
    return HIGH_BACK_EMF;
  else if ((1285 >= backEMFValue) && (backEMFValue >= 640))
    return MEDIUM_BACK_EMF;
  else if ((640 > backEMFValue) && (backEMFValue >= 400))
    return LOW_BACK_EMF;
  else
    return VERY_LOW_BACK_EMF;
}

void adjustableEchoRejectionStateMachine()
{
  unsigned long timeDifference = millis() -
    echoRejectionTime;
  if (timeDifference >= 50)
  {
    // Determine the current state
    int currentState = getBackEMFState(
    UMotorBackEMFValue_RollingMean);
    int maxEchoRejectionMicros=0, minEchoRejectionMicros=0,
    minValueFromBackEMF=0, maxValueFromBackEMF=0;

    // Use a switch-case to handle different states
    switch (currentState)
    {
    case VERY_HIGH_BACK_EMF:
      maxEchoRejectionMicros = 2400;
      minEchoRejectionMicros = 1800;
      minValueFromBackEMF = 2090;
      maxValueFromBackEMF = 3200;
      break;
    case HIGH_BACK_EMF:
```

```
316       maxEchoRejectionMicros = 3250;
317       minEchoRejectionMicros = 2400;
318       minValueFromBackEMF = 1285;
319       maxValueFromBackEMF = 2090;
320       break;
321     case MEDIUM_BACK_EMF:
322       maxEchoRejectionMicros = 7500;
323       minEchoRejectionMicros = 3250;
324       minValueFromBackEMF = 640;
325       maxValueFromBackEMF = 1285;
326       break;
327     case LOW_BACK_EMF:
328       maxEchoRejectionMicros = 17500;
329       minEchoRejectionMicros = 7500;
330       minValueFromBackEMF = 400;
331       maxValueFromBackEMF = 640;
332       break;
333     case VERY_LOW_BACK_EMF:
334       maxEchoRejectionMicros = 75000;
335       minEchoRejectionMicros = 17500;
336       minValueFromBackEMF = 0;
337       maxValueFromBackEMF = 400;
338       break;
339     }
340     // The map function uses linear interpolation to find
      corresponding echoRejection time according the back EMF
       speed
341     echoRejection_CurrentReadValue = map(
      UMotorBackEMFValue_RollingMean, minValueFromBackEMF,
      maxValueFromBackEMF, maxEchoRejectionMicros,
      minEchoRejectionMicros);
342     // To smoothen the echoRejection delay an weighted mean
      with focus on the old value is used.
343     echoRejectionDelay = weightedAverage(
      echoRejection_LastReadValue,
      echoRejection_CurrentReadValue, weightLast, weightNew);
344     echoRejection_LastReadValue =
      echoRejection_CurrentReadValue;
345     echoRejectionTime = millis();
346   }
347 }
```

Listing 7.2 Arduino class ComparatorRefHandler used for the reference signal to the comparator and voltage ripple counting