2 Software design

All of the software used to control the experiments or manually move the motors was written during this project. There are two main programs running on the Raspberry Pi during operation. One is communicating with the sensors and the motors (backend) and the other is communicating with the user (frontend). This separation allowed two different programming languages to be used for the two programs, so the strengths of each language could be utilised.

2.1 Backend

The backend program is written in the C++ language. This was chosen for two reasons, the first is the inherent speed of the language as it is a compiled language. This allows the program to run at high speeds even on such relatively weak hardware as the Raspberry Pi. This enabled higher sampling frequencies to be used and finer control of the motors. The other reason for choosing this language was the availability of the official motor control library created by the manufacturers of the motors. This library makes interfacing with the motors a much simpler task by creating an abstraction layer. This means that in the backend program simple commands can be used to control the motors, which are then translated to the more complex low-level instructions by the library.

The backend program has two main parts, first a setup part that runs once on startup, then a loop that runs until the program is stopped. The setup part establishes the connection with the motors, sets up the secondary encoder, initialises the message reader and creates the file that is used to save all the measured data. The loop part has three main tasks, it collects measurements, sends commands to the motors and saves the collected data to the output file.

2.1.1 Motor setup

The motor setup utilises functions from the aforementioned motor library. Initially, communication is established with the motors using the CAN bus adapter. Once a stable link is created, the PID parameters, the maximum torque, the maximum speed, the maximum acceleration, the internal loop frequency, and the maximum current draw are set. Most of the values used here were determined before this project and were not changed.

2.1.2 Secondary encoder

The encoder on the secondary shaft is connected directly to the Raspberry Pi using the GPIO headers. The encoder is an incremental quadratic type, which means it has two outputs (A and B) which change as shown in Figure 2.1. At each encoder tick (corresponds to 1/1024 of a full rotation of the encoder, equal to 1/(1024*4) of a full rotation of the secondary shaft) one of the two signals changes. By treating the encoder as a two-bit state machine a state transition table can be created as shown in Table 2.1.

During this part of the program, interrupts are attached to the two encoder pins. These interrupts call the same function (`encoder_callback()`) if one of the signals changes in either direction.

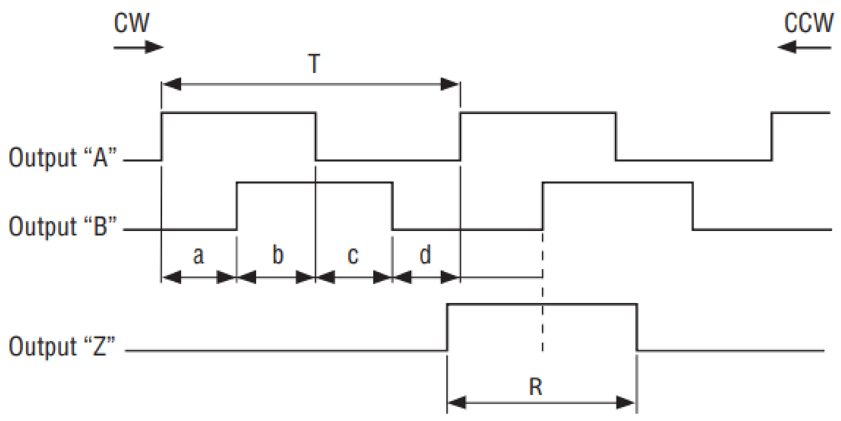


Figure 2.1: The output waveforms of the encoder, Figure from [3]

A global variable called `output_encoder_state` is also created, this is a four-bit number that represents the current and the previous state of the encoder signals in the format (in binary) $A_{n-1}B_{n-1}A_nB_n$.

A lookup table (`state_transtion_matrix`) is also created from the state transition table shown in Table 2.1. It has 16 elements corresponding to each possible transition, the positive transitions have a value of +1, the negative transitions have a value of -1 and the entries corresponding to no movement or error have a value of 0. After creating the table, its values get scaled by 360/(1024*4) to get the result in degrees.

Previous state ($A_{n-1}B_{n-1}$)	Current state (A_nB_n)	Movement	Value of `output_encoder_state`	Value in lookup table
00	00	None	$0000_2=0_{10}$	0
00	01	CCW	$0001_2=1_{10}$	+1
00	10	CW	$0010_2=2_{10}$	-1
00	11	Error	$0011_2=3_{10}$	0
01	00	CW	$0100_2=4_{10}$	-1
01	01	None	$0101_2=5_{10}$	0
01	10	Error	$0110_2=6_{10}$	0
01	11	CCW	$0111_2=7_{10}$	+1
10	00	CCW	$1000_2=8_{10}$	+1
10	01	Error	$1001_2=9_{10}$	0
10	10	None	$1010_2=10_{10}$	0
10	11	CW	$1011_2=11_{10}$	-1
11	00	Error	$1100_2=12_{10}$	0
11	01	CW	$1101_2=13_{10}$	-1
11	10	CCW	$1110_2=14_{10}$	+1
11	11	None	$1111_2=15_{10}$	0

Table 2.1: State transition table of the secondary encoder (Note that the actual values of the lookup table are scaled to give result in degrees)

When the callback function is called, `output_encoder_state` gets bit-shifted left by two bits and bitwise ended with 1100, effectively moving the "current" measurement to the location of the previous and deleting the previous. After the shift, the lower two bits get populated with the new values of the encoder output. The last step of the callback function is to add the value of the state transition matrix to the variable that stores the position of the secondary shaft. This is implemented by adding the `output_encoder_state`-th element of the `state_transition_matrix` array to the position variable.
Using these bitwise operations and the lookup table allows the callback function to be executed very quickly. This is important as this is an interrupt so the rest of the program is halted while this runs.

2.1.3 Message reader

During the setup phase a second thread is started. On this thread a loop is running that checks the specified named pipe for new commands. If no message is received, this thread waits until a message is sent by the frontend. When it detects a message, it reads it into memory and calls two functions to convert the raw text into usable commands. Each message has a keyword that specifies the action and, depending on the keyword, some numerical arguments delimited by spaces. The first function is `get_command()`, which extracts this keyword from the message. The second function (`get_command_vaues()`) is used to extract the arguments from the message. These extracted values then get stored in global variables that can be read by the main loop of the program running on the other thread.

2.1.4 Output file

As the last step of the setup, the log file is created. This .csv file stores all the data collected by the Raspberry Pi during the experiments. The name of this file is the current date and time. At this point, the first row is filled with the following columns: Time (HH:MM:SS), Milliseconds, Command, State, Drive position target, Drive ratio target, Drive position, Drive velocity, Drive current, Carriage position target, Carriage position, Carriage velocity, Carriage current, Output position, Output velocity. At each iteration of the main loop a new row is added to the file with the data.

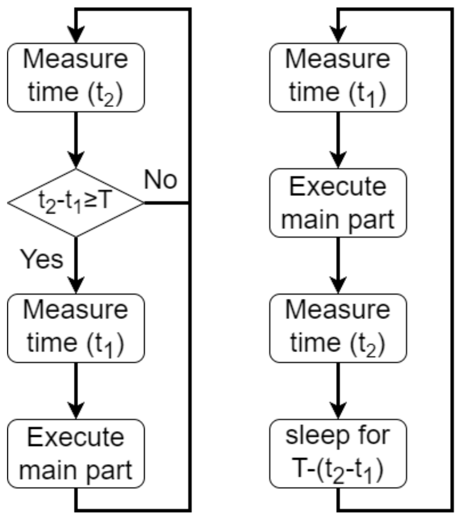


Figure 2.2: The two methods of timing the main loop

2.1.5 Timing of the loop

Two methods of timing the main loop were tested during this project as shown in Figure 2.2. The first method uses an infinite loop that measures the time since the last execution in every iteration. When the elapsed time exceeds the set time, the main part of the program is entered, where measurements are taken, the motors are controlled and data is saved. This approach would work well on systems without an OS, where only one program is running at a time. Here, where several other programs are trying to use the CPU at the same time, constantly checking for the time elapsed is not a good solution as it uses up resources and slows down other programs.

The second method also uses an infinite loop but instead of checking the time in every iteration, the thread is put to sleep after each iteration. To achieve accurate timings, the execution of the main part is measured and the sleep time is set, so that the thread sleeps for the remainder of the loop time. If the execution of the main part takes longer than the predefined loop time, an error is shown. Changing to the second option fixed problems which led to unexpected crashes of the program. No difference in timing accuracy could be measured between the two methods.

2.1.6 Collecting measurements

The first part of the loop deals with measurements, the `get_measurements` function is called. Within this function, the current time from the system clock is saved to provide a timestamp for the measurement. Then communication with the motors is initiated over the CAN bus to request the latest position, velocity and current measurement from the motors' internal systems. The received values are converted to degrees in the case of the drive motor and p-value in the case of the carriage motor. The received values are saved in variables that will later be used for control and logging. At this stage, a measurement message is prepared in the format specified in Section 2.3.2. The formatted message is then written to the named pipe for the frontend to read.

2.1.7 Executing commands

Two types of commands have been used during this project, a simple command that takes one iteration of the loop to execute and complex commands that take several loops. These commands are listed in Table 2.2.

Simple commands	**Complex commands**
`STOP` (0)	`INITIALISE_DRIVE` (0)
`CARRIAGE_GOTO` (1)	`INITIALISE_CARRIAGE` (0)
`DRIVE_GOTO` (1)	`STATIC_FRICTION` (0)
`CARRIAGE_SET_POS` (1)	`DYNAMIC_FRICTION` (0)
`DRIVE_SET_POS` (1)	`SPRING_CHARACTERISATION` (0)
`OUTPUT_SET_POS` (1)	
`PID_DRIVE` (4)	
`DRIVE_SINE` (4)	
`CARRIAGE_SINE` (4)	
`BOTH_SINE` (8)	

Table 2.2: The possible commands with the number of parameters in parentheses

`STOP`

When this command is received, the supply of the motors is removed so they come to a stop.

`CARRIAGE_GOTO` and `DRIVE_GOTO`

These commands can be used to move the carriage to a desired p-value and to move the primary shaft to a specified position. The arguments sent with these functions are the desired p-value and the final position in degrees. Both commands use the motor controller's built-in 'MotionMagic' controller. This generates a smooth velocity profile between the current position and the final position with predefined acceleration and maximum velocity values.

`CARRIAGE_SET_POS`, `DRIVE_SET_POS` and `OUTPUT_SET_POS`

These commands can be used to overwrite the stored encoder positions. These are useful after restarting the program, as the encoder positions are reset to 0 after each restart.

`DRIVE_PID`
The four parameters of this command: k_p, k_i, k_d and k_f. When this command is received the values are sent to the built-in motor controller to change the PID tuning of the drive motor. This was not used during the project.

`DRIVE_SINE`, `CARRIAGE_SINE` and `BOTH_SINE`
When this command is executed, the drive motor, the carriage motor or both are driven in a sinusoidal pattern with the specified amplitude (A), offset (θ_0), frequency (f) and phase (ϕ). The motion can be described by the following relationship:

$$\theta(t) = \theta_0 + A \sin(2\pi f t + \phi)$$

where t is the time of the system's clock.

The defining feature of complex commands is that their execution spans over several loops of the program and they have several execution stages. The `state` variable is utilised to keep track of the currently executed part of the command. The `posTarget` and `ratioTarget` variable is used to store a calculated target position or target torque (as a percentage of maximum torque) that is to be used during the next iteration. These variables are saved along with other data into the log file. A detailed explanation of the execution of the complex commands is included in their relevant sections.

2.1.8 Saving data

After the command has been executed, the next row of the log file is filled with data. Each row contains the timestamp recorded when the measurements were taken. The currently active command and its state are included as well as the other measured values listed in Section 2.1.4.

2.2 Frontend

The frontend is made using the Python language and the Tkinter GUI module. The language was selected because it is a high-level language that enables quick development. The main drawback of the language is its relatively slow execution time but this is not relevant for a user interface. The Tkinter module makes GUI development simpler by adding pre-built interface elements, such as buttons or textboxes. These can be configured and placed in the window to build up the user interface. The module also offers touchscreen compatibility, which is required here.

Visually the window is split vertically into two sections. The upper part contains the numpad and measured data display area. The lower part is used to control the apparatus with different commands split across the different tabs.

The frontend utilises two threads, just like the backend. The first thread is responsible for the received messages from the backend and the other thread handles the input from the user and sends it to the backend.

2.2.1 Numpad

As the Raspberry Pi is not connected to a keyboard, the only way to input numbers was the on-screen keyboard of the operating system. Using this proved to be cumbersome as this is a full keyboard that takes up most of the screen. During operation only numerical data needs to be entered, therefore a numpad was included in the user interface, which is always visible and enters numbers into the selected textbox. The numpad contains the numbers 0-9, a decimal point, a negative sign and a backspace key. Switching from the OS's keyboard to this numpad made using the GUI significantly faster.

2.2.2 Measurement display area

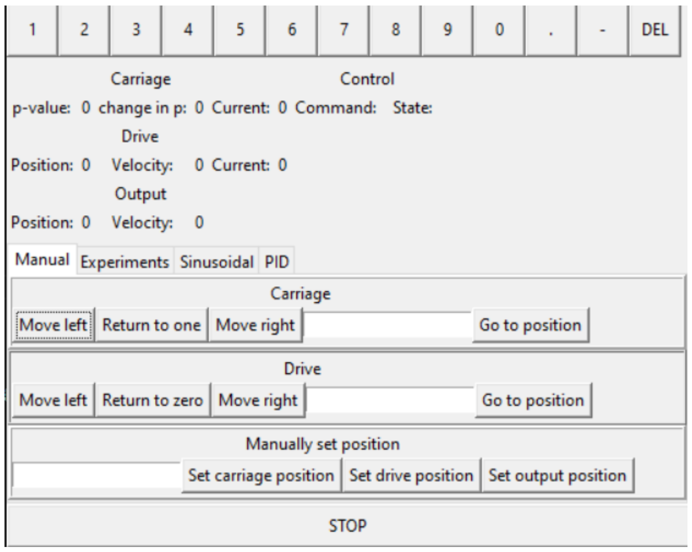
This area remains visible at all times and shows the position and velocity of all encoders, the current draw of both motors and the active command along with its state are also displayed. This area is updated using the first thread of the frontend, where an infinite loop is running, which checks for a message coming from the backend in each iteration. If no message is received it halts until a message appears. When a message is received it checks whether the format of the message is correct and raises an error if it isn't. In the next step, the data is extracted from the message and the text displayed in the measurement display area gets updated. The layout of the measurement display area is shown in Figure 2.3.

2.2.3 STOP button

A stop button is also visible on the GUI at all times. This can be used to quickly stop the movement of all motors.

2.2.4 Manual page

The first page that comes up when the program is started is the manual page shown in Figure 2.3. On this page both motors can be driven manually to a position. There are also buttons to move the driveshaft or the carriage a small amount (5° or 0.1 p) in either direction. On this page the measured position can be overwritten manually. This is necessary if the program is restarted, as the encoders can only measure the change in position, not absolute position. If the position is known it can be entered on the manual page, if it is not, the initialisation algorithms (Section 3) can be used to determine the position.



Carriage   Control
p-value: 0 change in p: 0 Current: 0 Command: State:
Drive
Position: 0 Velocity: 0 Current: 0
Output
Position: 0 Velocity: 0

Figure 2.3: The measurements display area with the manual page of the GUI

2.2.5 Experiments page

The experiments page of the GUI is shown in Figure 2.4. Each preprogrammed experiment and the initialisation algorithms can be started on this page. The dynamic friction experiment's parameters can also be entered on this page.

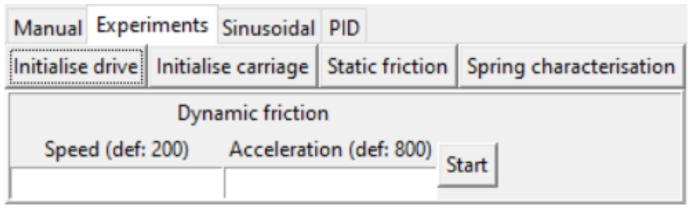


Figure 2.4: The experiments page of the GUI

2.2.6 Sinusoidal page

The parameters of the sinusoidal movement can be entered on this page.

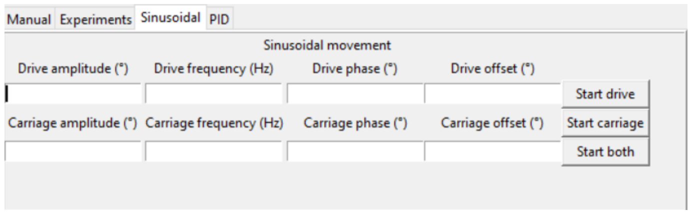


Figure 2.5: The sinusoidal page of the GUI

2.2.7 PID page

A page was created, where the PID parameters of the drive motor can be changed quickly. This was not used during the project as the parameters determined before this project were used.

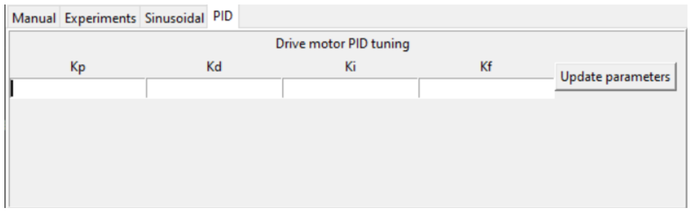


Figure 2.6: The PID setup page of the GUI

2.2.8 Sending commands

When a button is pressed on the GUI a command is written to the named pipe named `commands` in the format specified in Section 2.3.1. If the command requires parameters, an additional check is performed. The parameters are checked to make sure that they are valid numbers within specified limits.

2.2.9 Receiving measurements

The second thread of the frontend is reading the `measurements` named pipe. When a new message is sent by the backend, a function decodes the message and extracts the numerical values. When this is done, the values on the GUI are updated.

2.3 Communication

Communication between the two programs is accomplished using named pipes [7]. This is a built-in system found in Unix-like operating systems (such as the Raspberry Pi OS) that allows the creation of a pipeline that two separate programs can read simultaneously. These pipes can be read and written to just like regular files except that when a message on a named pipe is read, it is also removed from the pipe.

Two other methods were investigated: regular files and shared memory addresses. The main drawback of regular files is that only one process can open it at a time, so the reader would have to open the file and check for changes regularly. This leads to problems when the writer tries to send a message when the reader has the file open. Named pipes don't have this problem as both processes can open the file at the same time.

Shared memory addresses do offer higher data throughput but since the amount of data transmitted here is relatively low, this is greatly outweighed by the ease of setting up named pipes.

2.3.1 Commands

Commands are sent from the frontend to the backend. A single transmission always carries one command. The command consists of a keyword followed by some parameters (some keywords do not need parameters). The parameters are formatted as floating point numbers and they are separated from the keyword and the other parameters by spaces. A possible command message is shown in Figure 2.7. This command would cause the drive motor to move in a sinusoidal motion with an amplitude of 20°, a frequency of 1.4 Hz, a 45° phase and a 10° constant offset.

Keeping the keywords and the numbers in this human-readable format makes it possible to read these messages with other programs, which makes debugging easier. It also allows other programs to emulate the behaviour of the frontend by writing to this named pipe by hand. This way the backend can be used without the frontend.

```
DRIVE_SINE 20 1.4 45 10
```

Figure 2.7: An example command

2.3.2 Measurements

Measurements are sent from the backend to the frontend. Here a single transmission holds all the data measured at a timestamp. Each measurement is transmitted as a name-value pair separated by a space, the pairs are also separated by spaces. While transmitting the names with every transmission is not strictly necessary, (the decoder could decode the message without it, as the order of measurements is always the same) it is kept included so that the sent message can be read without a preprogrammed decoder. Including the names allows an extra check to be performed on the data to check its integrity which would be more difficult if the message only contained a list of numbers. Figure 2.8 shows a possible measurement message.

```
CARRIAGE_POSITION 1.43 CARRIAGE_VELOCITY 0 CARRIAGE_CURRENT 0.0
DRIVE_POSITION -12.3 DRIVE_VELOCITY  13.1 DRIVE_CURRENT 4.2
OUTPUT_POSITION 44.1 OUTPUT_VELOCITY 9.0 COMMAND DYNAMIC_FRICTION
STATE moving_positive
```

Figure 2.8: An example measurement message

3 Initialisation algorithms

All the encoders used on the apparatus are incremental encoders which means that they only measure a change in position, they cannot measure absolute position. Before an experiment can begin the current position of the shafts and the carriage must be entered through the interface's manual tab. If the position is not known, the following methods can be used to initialise the encoders.

3.1 Primary shaft

The primary shaft does not have any features that would allow a program to find the zero position automatically. Instead, an eyepiece with a line on it is mounted above the shaft. When this line is aligned with the line on the primary shaft, the shaft is at the 0° position. This eyepiece was added to the transmission system as a part of this project. This involved choosing the right eyepiece and designing the mount for it (shown in Figure 3.1). This mount was then manufactured by the technicians of the department.

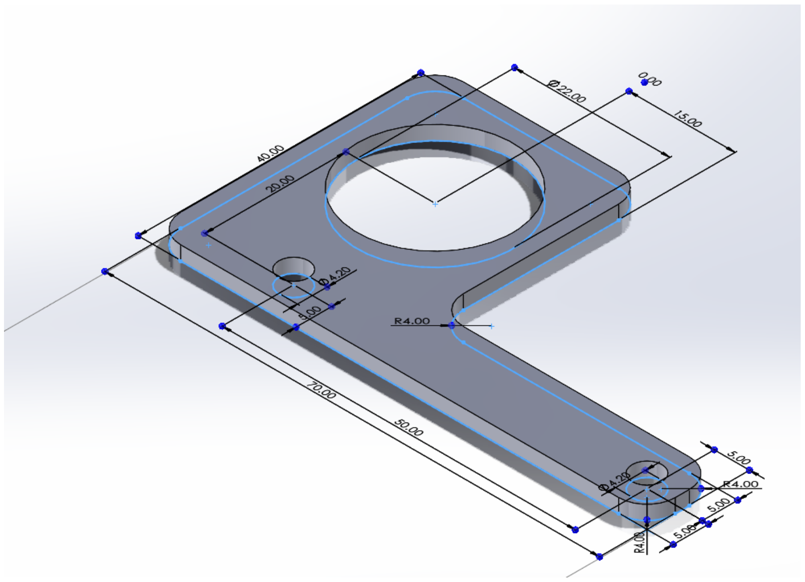


Figure 3.1: The CAD model of the eyepiece mount

3.2 Secondary shaft

If the absolute position of the secondary shaft is not known, it can be found using the endstops, the 0° position is defined to be halfway between the two endstops. This initialisation can be accomplished by a premade algorithm that can be started on the touchscreen interface. Initially, the system is slowly driven in the positive direction at a low torque level. In every loop the velocity is checked, if it falls to zero, it is assumed that the output shaft has hit the endstop. This position is saved and the drive is driven in the negative direction. When it stops again, the other endstop's position is saved and the zero position is calculated. Finally, the output shaft is moved back to the 0° position. The algorithm is shown diagrammatically in Figure 3.2.

The main difficulty encountered during the implementation of the algorithm was the tuning of the speed. If it is set too low, the shaft could get stopped before it actually reaches the endstop due to the changes in friction. On the other hand, if it is set to a too high value, the output shaft slams forcefully into the endstop which could damage the components.

3.3 Carriage

The carriage also lacks any easily identifiable features that could be used for initialisation. It does not have purpose-made endstops, its movement is only limited by the enclosure. The option of using this to find the absolute position was considered but eventually not used because neither part was designed to handle impacts like this. Repeatedly hitting the walls of the enclosure could damage the carriage or cause it to get out of alignment.

The position of the carriage is instead measured indirectly by measuring the p-value using the relative movement of the two shafts. At two different carriage positions the primary shaft is moved 50° from its starting position, the displacement of the secondary shaft is measured and thus the p-value can be calculated. Measuring the p-value at two different positions allows the change in p per encoder tick to be calculated using the fact that the p-value changes linearly with the carriage position, which is also directly proportional to the rotation of the carriage motor.

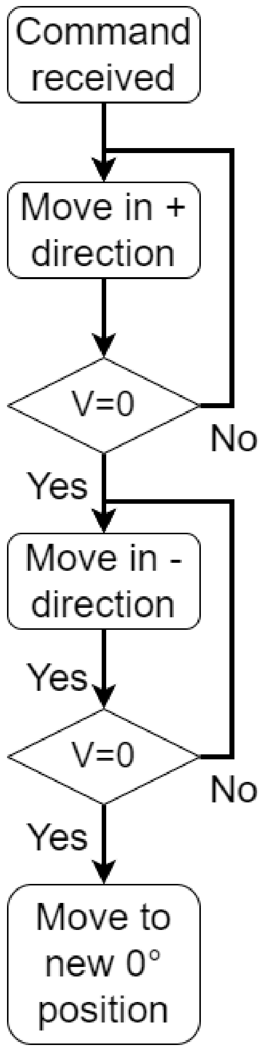


Figure 3.2: The initialisation algorithm of the secondary shaft