Project: Elevator System

MAS246

Group 5
Thomas Lønne Stiansen
Adrian Mathias Lervik Ling
Jan Laksesvela Haugstad

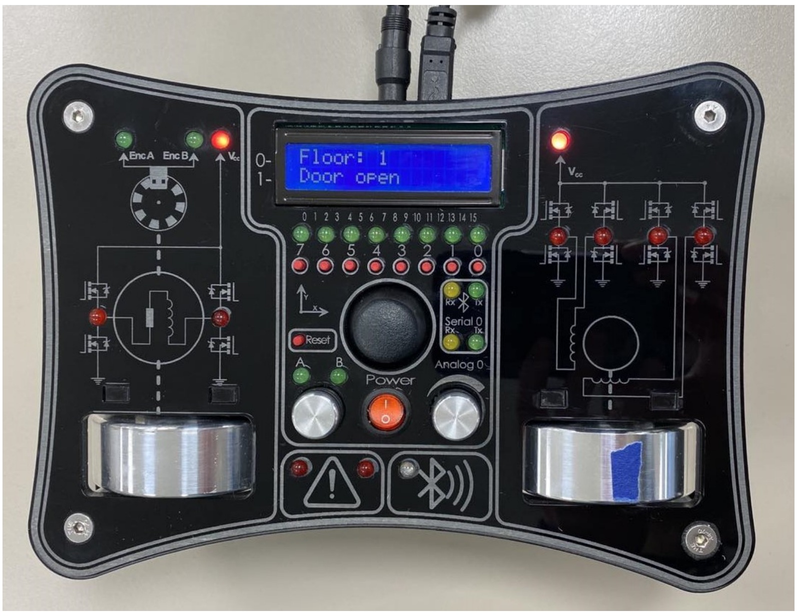


Figure 1: Servolab used in the project

Faculty of Engineering and Science
University of Agder
Grimstad, Norge
2023

Abstract

This project uses the all in one servolab provided by the faculty to simulate an elevator which consists of a single cabin which moves vertically. It utilizes the stepper motor to simulate the doors, the servo motor and its encoder to simulate the elevator movement, in addition to the LEDs and liquid crystal display as indicators of the elevators state. The simulated elevator takes inputs from the keyboard, in order to send signals for where the elevator should go. The project also consists of the different modes of a servo motor and the ways to read the signals from it's encoder, as well as the different control modes for the stepper motor. The project also includes the speed control of the elevator, which includes a PID-controller. Finally, a central part of the project is the software architecture and how more complex systems can be divided into smaller aspects.

Contents

1 Introduction

In the industry, the design process is central for a good product. The project focuses on the software architecture and different methods for getting a good overview of the system. The Unified Modelling Language, specifically class diagrams and state machine diagrams can be utilized in order to get a good grasp of how the system is supposed to operate. In more complex systems, isolation of the different functions is important so they don't interfere with each other in undesired ways; therefore the project addresses methods to achieve this division. PID-controllers are a central part of the industry, where its applications vary widely. Both PID-controllers and considerations regarding speed profiles are part of this project, as speed profiles also are important to consider in the pursuit of a design which fulfills the systems requirements. The industry requires a wide range of different sensors and actuators, where this project takes a delve into a few; namely the stepper motor, servo motor and its encoder, in addition to LEDs and a Liquid Crystal Display. As motors are crucial to the industry, it is important to be familiar with the different modes these operate with. This project explores the different ways to control the stepper motor and servo motor.

2 Theory

Stepper motor

The servolab utilizes a four lead two-phase stepper motor [1]. This stepper is a **"Hybrid Type"**, which is a combination of variable reluctance and permanent magnet motors. It consists of a permanent magnet in the middle, with rotor teeth on each end as shown in 2.1. This is connected with slightly less teeth on the stator, as shown in figure 2.2. These are spaced evenly around the circumference, so that the red teeth line fully up, the blue and green teeth line halfway up and the yellow teeth don't line up at all. This is an example with 50 teeth on the inside and 48 teeth on the outside, but the same principle applies to other hybrid stepper motors. The two pairs of each color are connected to sets of coils. When current is sent through these, the teeth are attracted or repelled to the adjacent teeth, depending on direction of the current. This results with a very precise movement, where one can magnetize a set of teeth to move the rotor a defined angle. This arrangement has a high torque with low angular velocity, which makes them suitable for applications where a short and controlled movement is required. Due to their high precision they are generally used in open-loop systems, where sensors are not required to check their position, since it is known as long as the stepper motor is operated within its rated load capacity.

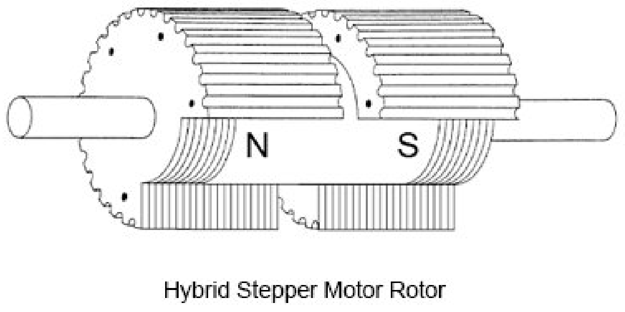


Figure 2.1: Hybrid Stepper Motor arrangement [2]

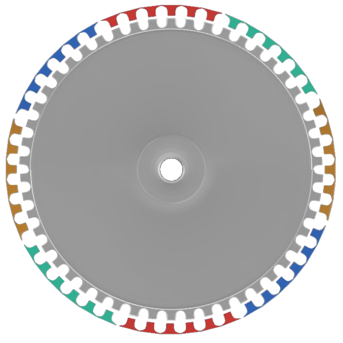


Figure 2.2: Setup of gear teeth in Hybrid Stepper Motor [4]

Hybrid stepper motors can be run in a couple of modes, which are defined by the way the coils are energized. A full step can be utilized where the system makes the teeth fully unaligned and unaligned with the corresponding energized coils. This can either be done with one-phase or two-phase; where the former uses one coil at a time to align the teeth directly, and the latter uses two coils at a time to align the teeth between the energized coils. Two-phase delivers more torque at the cost of more power used. Hybrid stepper motors can also be used with half-step, where it combines the two methods mentioned previously. It starts fully aligned with the energized coil, then energizes the next coils simultaneously in order to align them between the two. Then it can repeat the process with the next coil by disengaging the first coil, effectively pairing the teeth with the new coil. Similarly, hybrid stepper motors can also be used with microsteps, further increasing the accuracy. Stepper motors generally tend to get warmer when idle, due to current being supplied in order to keep them stationary in the same step, regardless of load as long as it's within its ratings. When moving it experiences a back EMF (Electromotive Force) in the opposite direction, which results in a lower voltage drop, therefore less dissipated heat.

Servo DC motor

The servolab also utilizes a geared DC servo motor with an encoder [1]. A DC motor is designed for continuous movement, where current is being sent into the rotor. The motor consists of a stator with permanent magnets, where the rotor will then align with the corresponding poles of the stator when its coils are being magnetized. To keep the motor going, there are a multitude of different solutions to reverse the current so it will go from that alignment to the next, such as brushes and commutators.

A servo differs from a regular DC motor in the sense that it is designed for having position control, where it is designed for moving to a specific position. The DC motor is combined with a driver, which can control the motor with Pulse Width Modulation (PWM) signals. Different drivers have varying interfaces, so the actual configuration depends on the type, therefore it is necessary to read the datasheet in order to set it up correctly. For the servo motor, the servolab utilizes a DRV8840 motor driver accompanied with a ACS712ELCTR-05B-T Hall Effect-Based Linear Current Sensor [1]. This driver interface includes three modes of running for the servo motor, which can be set by configuring the phase pin, decay pin and enable pin. The first mode is ***unipolar driving with slow decay***, which has a linear relation between PWM and voltage from 0 [V] to 12 [V] into the servo. It also has ***unipolar driving with fast decay***, which has an nonlinear relationship between the PWM and voltage in the same range. The last mode is ***bipolar driving***, which has a linear relationship between PWM and voltage, but this mode goes from -12 [V] to +12 [V]. The bipolar mode has the benefit of only requiring one signal to control the motor in both directions. The other two modes have higher resolution, but needs another signal to change the direction. When the PWM signal is equal to 0 [bits], the fast decay disconnects the motor and allows coasting, where it doesn't experience any electrical forces. When the PWM signal is 0 [bits] for slow decay, it results in the motor being shorted, which makes it brake with the back EMF (Electromotive Force).
The PWM signal that is being sent into the driver is proportional to the speed of the servo. A servo is usually accompanied with a sensor of sorts, so that its position is known.

Servo Encoder

The DC-servo motor in the servolab is equipped with an encoder, which consists of two Hall Effect sensors which senses the fields of a rotating disc with a magnetically polarized pattern [1]. One can choose to read in one of three ways: rising edge of only one signal line, resulting in 16 counts per rotation, or both rising and falling edge of one signal line, resulting in 32 counts per rotation, or both rising and falling edge of both signal lines, resulting in 64 counts per rotation. Even when using only the edges of one signal line, it is necessary to compare it to the second signal line in order to know the direction. As show in figure 2.3, when a rising or falling edge is encountered in signal 1, if the signal 1 is equal to signal 2 then the encoder is moving in the counter clockwise direction.

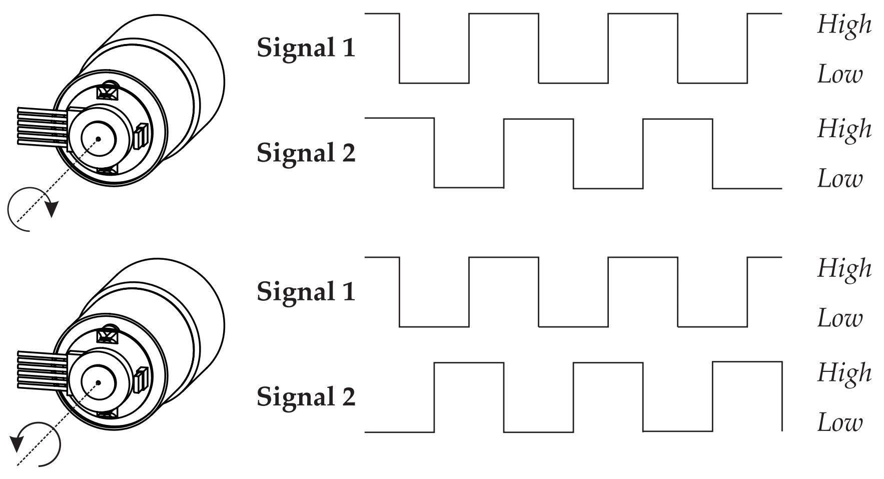


Figure 2.3: DC servo motor encoder signals

PID-controller

PID-controllers are widely used in the industry, where it has a broad range of applications. PID stands for "Proportional-Integral-Derivative", and refers to the way it works. It takes in the difference between the desired value and the actual value, which is called error $e(t)$, and sends out a signal that is related to that difference. The relation is shown in equation 2.1.

$$G_{PID}(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(t)dt + K_D \cdot \frac{de(t)}{dt} \tag{2.1}$$

The first constant K_P scales the difference so it has proportional relation between the error and the output. The second constant K_I relates the integral of the error and the output. It is known for eliminating the static error, where it will keep integrating, therefore increasing the term until desired value. The final coefficient K_D is for the derivative-term, which relates the slope of the error to the output. This is known for damping the output, as it will have a negative value when the output goes towards the setpoint, which is proportional to the rate of change. The derivative term should be used with caution, as it can make the system unstable, especially when there is noise in the signal. Therefore it can be accompanied with filters, such as lowpass filters or moving average filters.

Arduino inputs

In the Arduino IDE, there are a multitude of ways to take in and out signals, ranging from analog sensors to digital buttons. Physical buttons can be sent into the program with a built in function digitalRead(), where it will update the programs variable equal to the value of the button. Analog signals can be updated in the program with the built in function analogRead() with a maximum value of 1023 (as a result of 10-bit for the Digital to Analog Converter). One can also choose to take in signals from the keyboard using the serial monitor. Inputs from the serial monitor first set a value to Serial.available() that can then be checked if there is a pending input from the monitor, if there is no pending input then Serial.available() will be -1. If there is a pending input then the value can then be read using Serial.read() where the keyboard button is output as a byte in the ASCII format; if multiple buttons are sent in then there will be a queue of incoming bytes to be read one after another.

Unified Modeling Language (UML)

UML is a visual language used to describe a system. UML has multiple types of diagrams, two of which are relevant for the project; the class diagram and the state machine diagram. The purpose of UML is to have a generalized way to express code, which is applicable for all programming languages. It shall leave no room for interpretation, and make it so any person with the qualifications should be able to create the fully defined program from seeing the diagrams.

The ***class diagram*** gives a method of visualizing an object-oriented system by separating the system into classes which are displayed as blocks in UML, then showing how these classes interact with each other. The UML class diagram also includes abstract and interface classes, which are classes without strictly defined functions, which are used to create other classes that then define the functions which were declared in the abstract classes. The UML class blocks includes a name for the class, the variables (attributes) used, and the functions (methods) which they contain. Interactions with classes are displayed through arrows or lines from one class to another, and the type of arrow explains what type of interaction it is.

The ***state machine diagram*** works similarly to a flow chart but is directly connected to the system's class diagram. It describes the behaviour of the system and the states, but does not specify specifically how the behaviour is achieved.

3 Method

3.1 System Design

3.1.1 Version Control

Version control has been used throughout the project, using github for remote repositories. This has served the purpose of backups, and also the ability to revert to previous versions in case of undesired implementations.

3.1.2 System Requirements

In addition to the set of rules that were given out for the project, the group made a slide with lists of the system requirements, as well as the physical definition of the system. This was set as a starting point for the UML diagrams, which were done later.

3.1.3 UML diagrams

The system of the elevator was then set up with UML using class diagrams and state machine diagrams. To do this, the project used UMLet, which is an addon for Visual Studio Code. The class diagram explains the internal setup of the code which is split up into 5 Arduino .ino files. This consists of one main file; each of the rest are dedicated to a specific aspect of the elevator where each class/block includes the file's used variables and functions. Although the class diagram is not perfectly accurate with the actual setup of the elevator software due to limitations with the Arduino language, it explains the software's intended setup adequately. The blocks were inspired by the mechatronics concepts of sense, think, and act; therefore the elevator's system was split into sensors, Human Machine Interface (HMI) and actuators. The servo and stepper motor were split into each their own blocks, due to the fundamental difference between the two.

A state machine diagram was made as well, in order to have an overview of how the system operates and the different states of the elevator. It was split into a couple main states; prepare up and down, idle and the moving algorithm. The moving algorithm was then split further into sub-states, as it is a more complex state.

3.2 Elevator function

This section will explain briefly how the system is set up to work, but the details of the system's architecture is discussed in chapter 4.

3.2.1 Physical Connection with servolab

The units of the servolab were converted in the software to SI-units to better simulate the elevator's movement rather than using rotations of the motors. As the elevator is not connected to a physical drive, one rotation of the servo motor was decided to be 1 [m] of the elevator's vertical movement where each of the floors are 5 [m] apart. The group decided that one rotation of the stepper motor would close the door, and therefore the stepper runs 200 steps in the corresponding direction depending on if the door should open or close. The elevator was designed to have 4 floors, although the created system allowed the potential to expand for more floors with ease.

A requirement was set to have the door closed when moving to protect the users of the elevator. To close the doors and move the elevator to the next floor, the stepper motor then had had to do 1 rotation, and the servo had to do 5 rotations.

3.2.2 Elevator Mock-up

Before the system was created with physical sensors and actuators, the system was designed with simulations. This mock-up is further discussed in 4.2.1.

3.2.3 PID-tuning

The PID was tuned manually by first setting the gain to a low value and increasing it until the setpoint was reached at a desired speed. A PID-controller was utilized for multiple reasons, but one of the most important being the fact that it ensures that the elevator reaches the desired floor without a vertical gap. Since there is no static error in the system, the setpoint will be reached without needing an integrator term; this is discussed further in 6.5.2. The derivative term was then enabled to control the speed of the curve further, this allowed the group to control the deceleration to an extent as the elevator approaches the desired floor.

4 Hardware and software architecture

4.1 Hardware

Table 4.1: Mapping of used components

	Hardware	Pins	Features/Comments
Sensors(IN)	Servo Encoder	20, 21	pin 20 = signal 1, pin 21 = signal 2. Using 32 counts per rotation which equals 4192 counts per rotation due to the servo's 131:1 gearbox
	Keyboard buttons	-	Serial input, buttons 1-4 for interior cabin buttons, buttons q,w,e for outside up buttons, buttons s,d,f for outside down buttons
Actuators(OUT)	Servo motor	7,6,5	Pin 7 = enable, Pin 6 = phase, Pin 5 = decay, using 131:1 gearbox and slow decay mode
	Stepper motor	69, 68, 67, 66	200 steps per rotation, Pin 69 = A, pin 68 = A phase, pin 67 = B, pin 66 = B phase, using DAC.h library
	LCD screen	41,40,37,36,35,34	Using liquidCrystal.h library
	LEDs	49-46	pin 49 = LED 0, pin 46 = LED 3

The group chose to exclusively use inputs from the keyboard rather than the physical buttons on the servo-lab unit as it gave the software greater flexibility and made the HMI code more organized. A decision was made to use the keyboard numbers 1 to 4 as the internal elevator controls, and the two rows of keys underneath the numbers on the keyboard as the corresponding outside buttons for calling the elevator. This means the keys from "Q" to "R" are the up-buttons and from "S" to "F" are the down-buttons on the "outside of the elevator". Since the bottom and top floor only have one direction, the bottom floor only has the button "Q" as the up button, and the top floor only has the button "F" as the down button. The system uses LEDs to display which floors have been requested, and an LCD to display the current floor to the user of the elevator. As a part of the servolab, the project utilizes the stepper motor and the servo motor along with its encoder.

4.2 Software Architecture

The software architecture was based on the UML diagrams, described in 3.1.3. The resulting diagrams are shown in A.1.2 and A.1.3, and further discussed in 6.1. The following subsections in this section go into detail of how each .ino file block in the UML class diagram was set up code-wise.

4.2.1 Elevator Mock-Up

The creation of a mock-up which does not drive the motors but only simulates the system in the serial monitor was suggested, serving as a skeleton project. This skeleton project was divided into individual programs (.ino files) for each of the elevators aspects, then later put together to one project. This made debugging more effective and focused, as temporary changes could be made to one function's code without having it affect other functions. While developing, one could test each aspect of the elevator by providing inputs and observing if the outputs were as expected.

An elevator project is an example of a project where object-oriented programming (OOP) could be used with great success, however the Arduino language is based on C and does not have classes and consequently lacking the ability to use OOP. Because of this, the group used multiple files to simulate how classes would be used in a language like C++. Variables which would be used across the .ino file were placed into namespaces as opposed to using global variables; however due to the way the Arduino compiler works, the namespaces could not always be in the same file that corresponded to it and had to be placed in the main .ino file to be accessed from other .ino files. The namespaces were named accordingly to their usage such as "servoVars" if the namespace were variables which were primarily used for the servo motor to show which file the variables were supposed to be used for.
The elevator software does not have hardcoded functions for the induvidual floors with the exception of the top floor and outputs for the specific keyboard buttons; this combined with the use of structs for the keyboard inputs allows for flexibility with choosing how many floors the elevator has by simply setting the maximum floor and adding the new floor input buttons.

4.2.2 Main file

The main file of the elevator is based on the UML state machine diagram and uses switch cases along with an enumerator for the states to achieve the described functionality of the state machine diagram.
The elevator can only move to the next state once the exit requirement is reached. Some functions also run continuously every loop despite the state, such as checking the buttons.

4.2.3 HMI and Queue System

To check the buttons, polling was used where it checks for the signals each loop. The button inputs from the serial monitor are checked in a switch case which then outputs a struct including the floor number and the direction; inputs from inside the elevator cabin are given a separate output for the direction which states that the direction for this input is irrelevant.

The queue system was placed inside the HMI file, and consists of two separate sub-queues, a main and an alternative queue. The main queue contains the requested floors which are in the same direction as the elevator is moving in, and the alternative queue contains the requested floors which require the elevator to change direction to reach. When all the requests in the main queue are cleared, the alternative queue becomes the main queue and the elevator then continues the movement through the new main queue which is most likely in the opposite direction. This system was based on the requirements for the elevator, which states *"An elevator can only change direction, if there are no requests to serve in its direction, but there are other requests in the opposite direction."* [3]. If a new floor is requested in the same direction as the elevator, the request can just be put at the end or start of the main queue depending on the floor number, rather than using a single queue and keeping track of an index of where the floors which are in the opposite direction of travel are, which makes adding newly requested floors to the queue more complex.

The HMI includes two kinds of visual outputs; LEDs for the queued floors and an LCD which displays the status of the door, the current floor if idle, or direction and the floor the elevator is moving to if the elevator is in motion. The function which enables the LEDs first checks all the requested floors in both queues, and will set a bit in a byte which corresponds to the floor number, for example the LSB of this byte corresponds to LED 0 and the MSB would refer to LED 7. One LED was used for each floor on the servolab, for all the floors from LED number 0 to 3. The code then checks with bit shifts and boolean operations if the bit corresponding to the LED is 1 or 0, if it is 1 then the LED turns on and if it is 0 then the LED turns off. The LCD uses one of two functions, which has separate codes depending on the number of inputs; this way the LCD can be written to with one function, even when having a dynamic amount of inputs. The function also takes in the height of the LCD, which can either be 0 or 1, 0 for the top LCD output and 1 for the bottom.

4.2.4 Stepper Motor

Since the stepper motor has to be run with 4 separately defined inputs for a full step, the code for the stepper was optimized so the steps could easily be reversible and to be as efficient as possible for smooth movement. The configuration was inspired by the presentation provided [5]. The stepper motor code uses the boolean operator AND (&) with 0000 0011 and an unsigned integer. It increments or decrements depending on the direction in every step, where the output of each step is defined in a switch case inside the loop. The switch case has four possibilities ranging from 0 to 3, as a result of the two bits used for the boolean comparison, where the binary and decimal forms are equivalent. Since unsigned integers have a defined behavior on overflow, only an 8-bit unsigned integer was used to save space. This results with the value going back to its original state when it goes beyond 255 (maximum value for 8 bits: $2^8 - 1$). Also before that when it goes above 3 (equal to the binary value 0000 0011) it will increment the next digit, which is irrelevant for the boolean operation, resulting with the steps looping nonetheless. This is shown in 4.2.4.1.

4.2.4.1 Illustrative pseudocodes:

```
compare = 0000 0011
base = 1111 0000
output = base AND compare
```

When two bytes are ”ANDed” together, each relative bit are compared. This results with *output* being equal to 0000 0000, since the last two digits of *base* are 00. Therefore the first step "0" for the motor runs, and after this *base* increments for the next iteration:

```
base++          // Resulting with base = 1111 0001
output = base AND compare
```

This results with *output* being equal to 0000 0001, since the last two digits compared are 11 and 01. That means the second step "1" runs, thereafter it increments again. The first 6 leftmost (most-significant-bit) values of *base* are irrelevant since they are ”ANDing” with zeros. This function is reversible since when *base* goes down by 1, the previous step will be ran even with overflow:

```
base = 0000 0000
output = base AND compare
```

This results with *output* being equal to 0000 0000, and the first step "0" runs due to *base* ending with 00. When *base* decrements by one, it leads to *base* overflowing in the next iteration and beginning from the maximum value:

```
base--          // Resulting with base = 1111 1111
output = base AND compare
```

This results with *output* being equal to 0000 0011, which means the last two digits 11 dictate the fourth step "3", which will move the stepper motor in the opposite direction.

This method only works for looping through numbers of the multiple of 2, in which the number of loops depends on how many bits from the LSB are set to 1 on *compare*.

4.2.5 Servo Motor and PID

The servo motor was set to slow decay mode by setting the decay on pin 5 to low; this allows the motor speed to be controlled with PWM on the enable pin and the direction on the phase pin, where the motor runs counter-clockwise when the phase pin is low, and clockwise when high. The configuration for the slow decay mode was inspired by the presentation [5].

The group set a requirement for the maximum starting acceleration of the elevator to be 1.2 $\left[\frac{m}{s^2}\right]$ and a maximum speed of 1 $\left[\frac{m}{s}\right]$. This value was set arbitrarily and was implemented as a variable, so it could be changed if used for an actual elevator. In order to achieve a trapezoidal speed profile with the requirements stated above, the servo motor was designed with two states: constant acceleration to maximum speed, thereafter PID-control with a saturation for the maximum speed.

These states were created rather than using only PID as the PID controller gives a very sharp initial acceleration due to the large error. The PID calculations also includes the deadzone for the motor, where the deadzone is negative if moving down and positive if moving up, but uses the same value.

The servo motor code cannot have delays in its function as delays stop the entire CPU proccess until the time has passed, therefore a method with logic using the built in Arduino Millis() function was used, which gives the amount of milliseconds passed since the program was first started; this was then converted to seconds for calculating the dt for use in PID calculations. If delay functions are used, this will halt the entire CPU process, resulting in interrupts being queued. During this delay, the system will continue with the same values, resulting in the motor spinning with a constant value while the delay is active and not making new motor calculations.

A threshold for when the elevator is close enough to the desired floor was added; when it reaches this a timer begins. The servo motor will keep using PID-control to stay close to the desired height, and when the timer reaches 2 [seconds], the servo motor stops (as long as it is still inside the threshold). This gives the servo motor a smooth transition for when the elevator goes from moving to stationary.

4.2.6 Sensors

The servo motor encoder has the option to give a maximum of 64 counts per rotation, however due to the gearbox on the servo motor this count is multiplied by 131. Therefore the group decided to use the option with 32 counts per rotation by reading the rising and falling edge of one signal line which was considered adequate, as it gives the encoder an accuracy of:

$$\frac{N \cdot CPR\left[\frac{counts}{rotation}\right]}{360[degrees]} = \frac{131 \cdot 32\left[\frac{counts}{rotation}\right]}{360[degrees]} = 11.6444\left[\frac{counts}{degree}\right]$$

The code for the encoder was inspired by [6] and includes an interrupt which enables on rising or falling edge of signal 1 of the encoder; once this interrupt is made then a function is run which checks if signal 1 is the same as signal 2, if true then the encoder has turned counter-clockwise, and if false then the encoder has turned clockwise. Depending on this output, a counter then increments or decrements. A function is used to get the value of this counter by disabling interrupts before getting the counter value then enabling it again to prevent errors if the counter were to shift while being read.

5 Results

5.1 System Requirements

The slide for the system requirements is shown in A.1, the class diagram is shown in A.1.2, while the state machine diagram is shown in A.1.3.

5.2 Elevator movement

The stepper motor was chosen to run with 5 [ms] between each step.

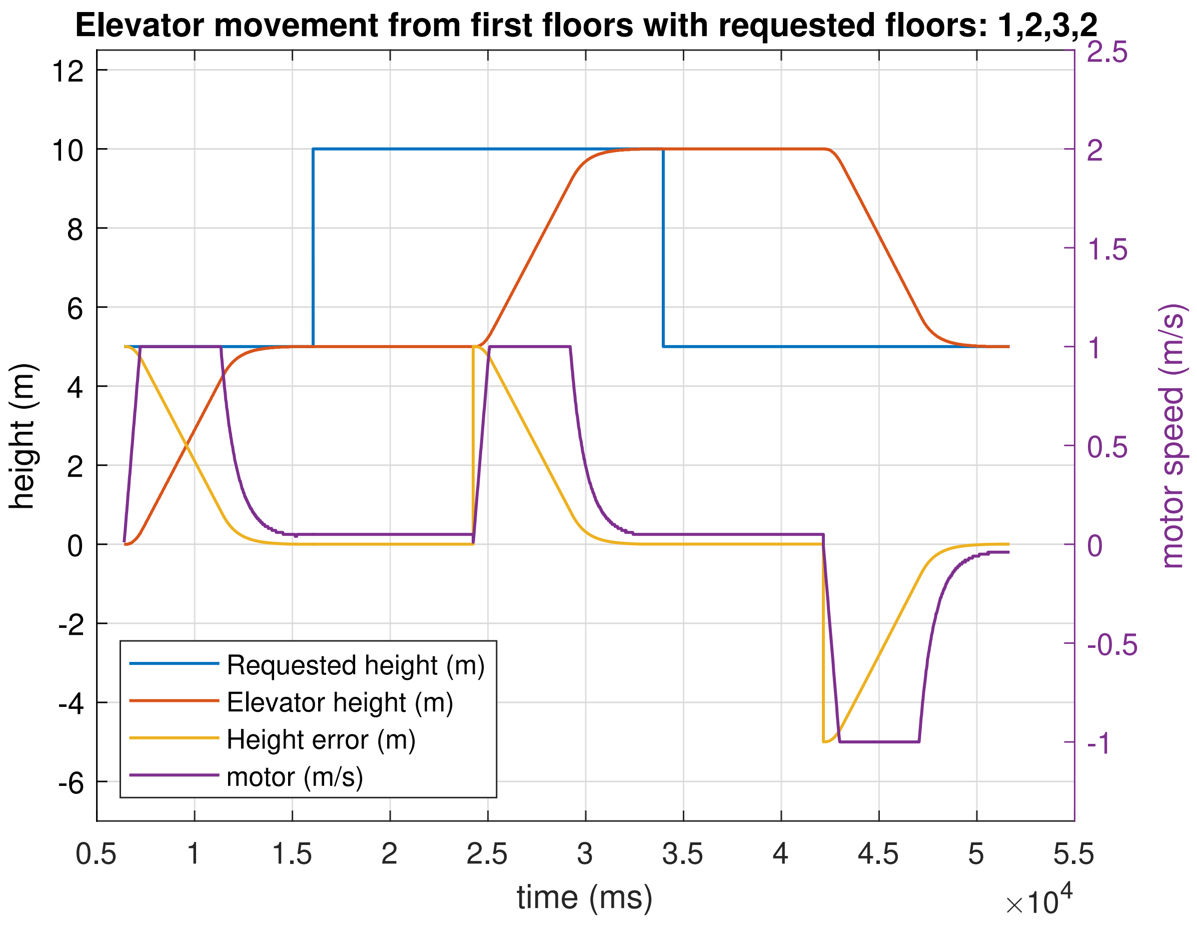


Figure 5.1: Elevator movement with the following floor requests: 1, 2, 3, 2

The graph 5.1 shows how the elevator reacts to a queue of floors with directional changes. There is a delay between each floor change which comes from the doors opening, waiting 2 seconds, then closing again.

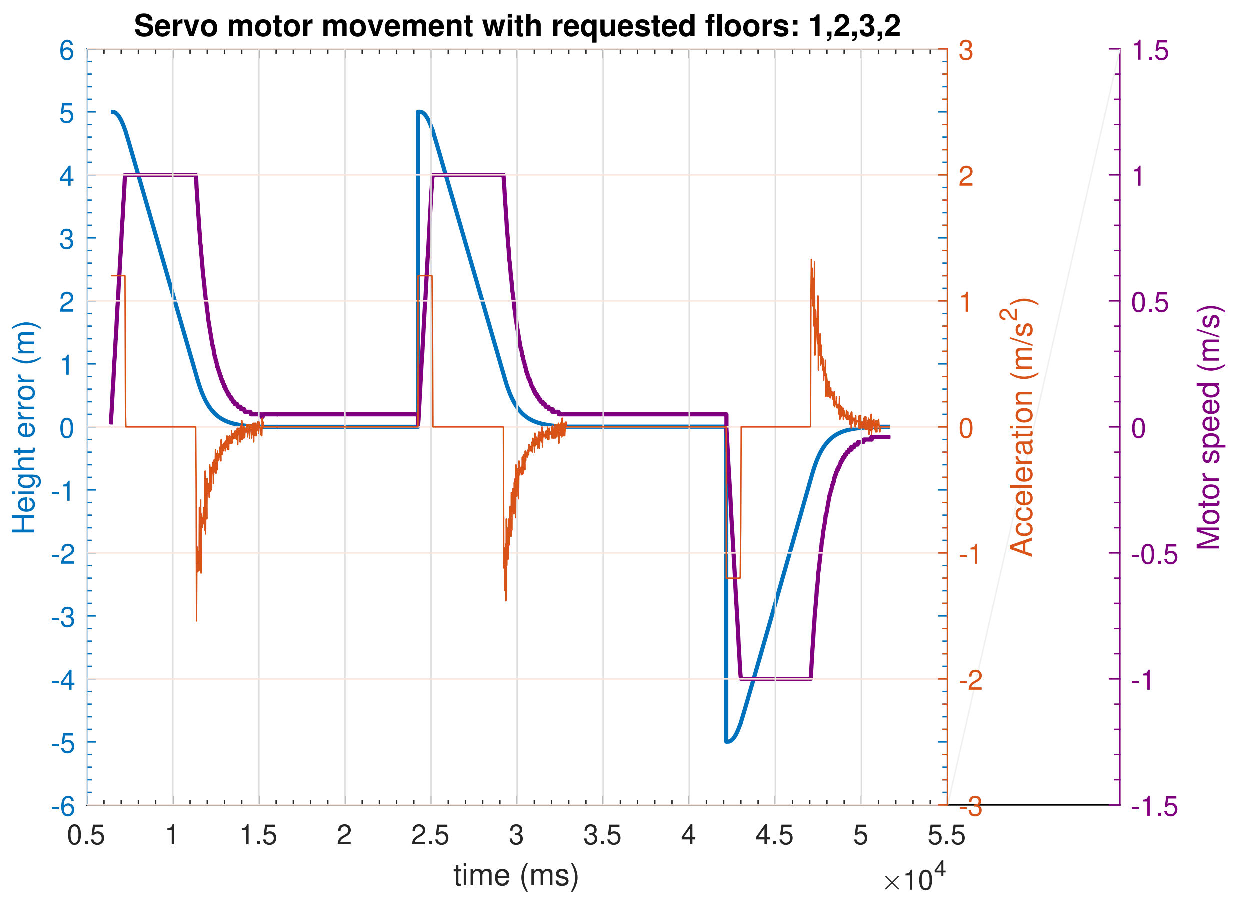


Figure 5.2: Angular velocity and acceleration of servo

Figure 5.2 shows the speed of the motor, as well as the acceleration and speed of the elevator during the same move requests as figure 5.1. Figure 5.2 displays the relationship between the acceleration and speed, where the speed goes up until it reaches the saturation at 1 $[\frac{m}{s}]$ from the DC-motor function.

The elevator reaches the desired floor with a gap of 0.00167 to 0.00429[m] or 1.69 to 4.29[mm], and has a maximum starting acceleration of 1.2 $[\frac{m}{s^2}]$ and a maximum deceleration of ± 1.6 $[\frac{m}{s^2}]$ with the current PID-parameters as the elevator approaches the requested floor, as shown in 5.2.

5.3 Video

A video of the elevator is added as an attachment to the report and shows the functionality of the elevator.

5.4 HMI Displays

The LCD shows the state of the door on the second line and on the first line the LCD displays either what floor the elevator is on or what floor the elevator is moving to and in which direction depending on the system's state. The floor the elevator is moving towards changes dynamically as a new button is called. This is the display which the user of the elevator would see.
The LED displays the floors in the queues but the update of the LEDs after pressing a button is sometimes delayed depending on the state of the machine.
The serial monitor gives deeper information on the how the elevator is moving and what the next floors in the queue are.

5.5 PID-controller

The values for the PID parameters after manual tuning were:

$$K_P = 1.25 \qquad K_I = 0.0 \qquad K_D = 0.1$$

The servo motor also had a deadzone compensation of ± 0.043, the sign depends on the direction of the elevator. The testing was done with the values plotted in the serial monitor. This was done in order to keep the PID-tuning so that the deceleration did not exceed acceptable levels. The current PID-parameters resulted in the following deceleration values when going up and down:

$$\textbf{UP: } upDeceleration = -1.6\left[\frac{m}{s^2}\right] \qquad \textbf{DOWN: } downDeceleration = 1.6\left[\frac{m}{s^2}\right]$$

Compared to earth's gravity, these values correspond to:

$$\textbf{UP: } \frac{g + upDeceleration}{g} = \frac{9.80665\left[\frac{m}{s^2}\right] - 1.6\left[\frac{m}{s^2}\right]}{9.80665\left[\frac{m}{s^2}\right]} = 0.8368$$

$$\textbf{DOWN: } \frac{g + downDeceleration}{g} = \frac{9.80665\left[\frac{m}{s^2}\right] + 1.6\left[\frac{m}{s^2}\right]}{9.80665\left[\frac{m}{s^2}\right]} = 1.1632$$

6 Discussion

6.1 UML

6.1.1 Class diagram

The system was split into three sub-blocks, which reflect the mechatronics concepts of sense, think and act. The sensor-block and actuator-block represent sense and act respectively, and the HMI represent the think aspect because it includes the queueing system.

The arrows and lines in the class diagram represent the way methods or functions are called from one class to another, it also describes how classes interact with each other. The elevator.ino file which is the main file for the code, only calls functions from the HMI and the actuators, therefore the interaction was represented with a one way association arrow, where the arrow points towards the HMI and actuator classes. Both the servo and the stepper are represented as inheritances from the empty actuators class, which shows that servo and stepper are classes represented by the actuator class. The sensors.ino file does not interact with any other class other than the servo.ino file for the servo encoder, therefore a one way association arrow was used as the servo.ino class calls functions from sensors.ino.

Enumerations were made using the «Enumeration» identifier, and are attached to each declaration of the enumeration, the struct used for the button outputs were described in the same way, but since structs have public variables by default, a "+" sign was added to each property. The "-" sign which tells that a variable is private within a class was not used for the class UML as it might be incorrect because in the code the variables are not truly private like they would be in a class.

The variables used for the classes are placed in the property section of the classes, and the global variables are represented with a "+" which says that they are a public variable in the class; this form of notation is not exactly correct as the project doesn't use classes, but the group thought it was a good way to represent global variables because the variables are not within a namespace in the code. This unfortunately causes a discrepancy between the UML class diagram and the actual software since variables which were supposed to only be used for one file had to be placed in the main file due to trivial reasons such as outputting a value in the serial monitor in the main .ino file. The namespaces in the code have been given intuitive names corresponding to which file the variables are supposed to be accessed from to make up for this difference. The reason for dividing the systems in this manner, is to keep the system organized and restrict access to the variables so they are only available to their designated functions. This is especially important in more complex systems, as names which seem intuitive at the time may not be in the future, or for the rest of the team working on the project.

6.1.2 State machine diagram

The UML state machine diagram describes how the elevator works by separating each function into states which the elevator can be in. The state machine diagram was designed around the requirements for the elevator, such as the door requiring to be closed before continuing to the moving algorithm state. If the elevator is in a state and has not reached the requirements to move to the next state, then the elevator will continue looping the current state until it has. The state machine does not include all the used functions, as many functions call other functions internally, so the group decided to only include the primary functions which are often called from elevator.ino.

6.2 Software choices

Polling was used to check the buttons as opposed to using interrupts. Polling allows the system to check the buttons at specific points in time rather than randomly as the button is pressed. An interrupt during a bad time could reduce the smoothness of the system as when a button is pushed, the system uses a lot of processing for calculating the queue, and the inclusion of the serial monitor printer slows this down significantly. Interrupts affect the stepper motor the most and a serial print during the steppers movement would make it unstable. This could be fixed with disabling interrupts during important functions but the group decided that polling at a specific time was a better alternative.

Because the elevator is able to change the floors dynamically by checking serial inputs while the loop for the servo motor is running, the buttons are allowed to be pressed rapidly. This can cause a lot of the processing time to be spent on checking the button inputs instead of running the servo motor, which sometimes causes the PID to act unpredictably. This issue may be caused by the serial monitor's slow speed to output information, which could be solved by reading the buttons on a separate thread rather than increasing the baud rate of the serial monitor. One way to eliminate variable loop times all together is to have a constant loop time and run other calculations in parallel instead of everything in one main loop. This is especially important to consider for more complex systems which do a lot of calculations which take time.

6.3 Stepper and Servo Choices

6.3.1 Stepper Modes

This project utilizes full step for the stepper motor. There are a couple of reasons for this. The system does not require the precision of the half step or micro step, therefore the full step mode is adequate. Furthermore, the full step is more capable of stable operation at higher speeds than its divided counterparts. It also results in more torque, which is good for moving its load; the door.

6.3.2 Servo

The group decided to set up the servo using the ***unipolar driving with slow decay mode***. This is because it gives a linear relationship between the PWM output to the driver and the speed of the servo. In addition to the added simplicity compared to its fast decay counterpart, it also has the ability to brake the motor when the PWM is equal to 0, which is great for halting it in the desired position. Furthermore, it has greater resolution than the bipolar mode. A delay of 5[ms] between each step was chosen as it gave the door movement a good speed with consistent results.

The servo position was estimated using the encoder, where it reads the ***rising and falling edge of one signal line*** and compares it to the other signal line. Compared to reading both lines, which result in double the counts per rotation, interrupts are called half the amount of time. This results in a more smooth operation, as the interrupts halt the calculations which result with undesired behaviour as mentioned in 6.2. The increased resolution by reading the rising and falling edge of both signal lines is not needed. This is because the current resolution is multiplied by the gear ratio, as the servo motor shaft rotates that many times in order to move the load one full rotation.

As soon as the elevator reaches the current floor, the request is cleared which was against one of the elevator requirements which states: *"The requests at one floor are cleared once the door is fully open"* [3]. The requests clearing before the door is open is not an issue in the system because at the same time a request is cleared, the door has to open and close before the elevator can continue its functionality. The group interpreted the requirement to be stated in order to keep the timing of requests in mind when designing the queue system. Since it clears the request from the queue and forces the door movement before moving the elevator, it was considered appropriate. The code for the LEDs however, has another logic which follows this requirement and therefore turn off after the door opens.
The group also decided to use the terms "up", "down", and "idleMotor" instead of "Winding", "Unwinding", and "stopped" as it made the code more understandable during development, the functionality was not changed despite the name change.

6.4 Additional Feature Suggestions

The designed elevator lacks a few features that should be implemented for a real life elevator to run safely. Because there is no sensor on the door, it is impossible to check the position of the stepper motor. In the code, after the function that drives the stepper, the system assumes that the door has always closed or opened successfully which might not always be true in a real life scenario. Generally it is okay for steppers to be in open-loop control, but for commercial use, such as an elevator this may need to be considered. If a person got stuck in the door, the system should have security measures such as sensors which stop the door from trying to close if it measures resistance. Another concern is accessability options. For a person with reduced movement, options should be available; for instance the ability to use voice commands instead of buttons. Or a simple addition for blind people; braille dots beside each button.

6.5 Elevator Control

6.5.1 Speed Profile

The project utilizes a combination of constant acceleration, PID-control and saturation of motor speed. The PID-controller ensures the absence of vertical gaps between the elevator and the desired floor. The constant acceleration was done in order to keep the acceleration from going beyond comfortable levels, as acceleration is related to the force excerted on elevator and its users. Optimally, a trapezoidal speed profile is desired, where the elevators motor has a constant acceleration, then constant speed and then constant deceleration.

The design used in this project for the elevator was created by having a constant acceleration until the maximum speed was reached. When it reaches this threshold, the PID-controller takes over. The speed is being capped with a saturation at the maximum speed until the elevator is close enough to the desired floor so that the PID-controller gives out a lower value than the saturation such that the motor decelerates. If the PID-controller would have been used through the entire process, it would have given a strong acceleration in the beginning before reaching the constant speed, which is uncomfortable for the passengers. This is due to the control input being proportional to the difference between desired and actual height. This solution displays the use of constant acceleration and PID-control, which is adequate for a simple elevator simulation such as the servolab. The acceleration was measured for the current system to be maximum 0.8368 and 1.1632, as shown in 5.5. This means that the system is 16.32% lower and higher than earth's gravity, which was considered adequate.

For an actual elevator with variable loads (people, baggage, etc. going into the elevator) this would not have been good enough. This is due to the power sent into the motor being constant regardless of the load the system experiences, which results in different speeds at different loads. In order to achieve a trapezoidal speed profile with the same shape regardless of loads, a possible solution could have been cascade control such that the speed of the motor would follow the desired profile, and send out more power when more load was applied to achieve the same speed.

6.5.2 PID-tuning

An attempt at using the Zieglers-Nichols method was done for closed loop tuning, however due to the fast response time of the DC-motor, this method gave very high values especially for the K_I term, which did not work well on the elevator system. An explanation for this is that because the load is very low, the elevator appears to be an over stabilized system. The integrator term for the PID was not required since there is no static error in the system for the integrator term to compensate for. This could be noticed when even putting a low Kp term for the elevator, it would still reach the desired set point. In a realistic scenario with a load on the elevator, an integrator term would be more suitable as the load would create a static error.

6.5.3 Deadzone Compensation

The servo motor was observed to have a deadzone before turning, which was compensated for by adding a small constant to the output going into the driver. Deadzone compensation can also make PID-control simpler, as without can result in the PID-controller sending out a higher signal in order to compensate for the system's lack of response. It has been experienced in the past that the PID-controller tends to overshoot when passing this deadzone. When doing this with servomotors, it can lead to the PID oscillating around the desired position instead of decelerating in a controlled manner.

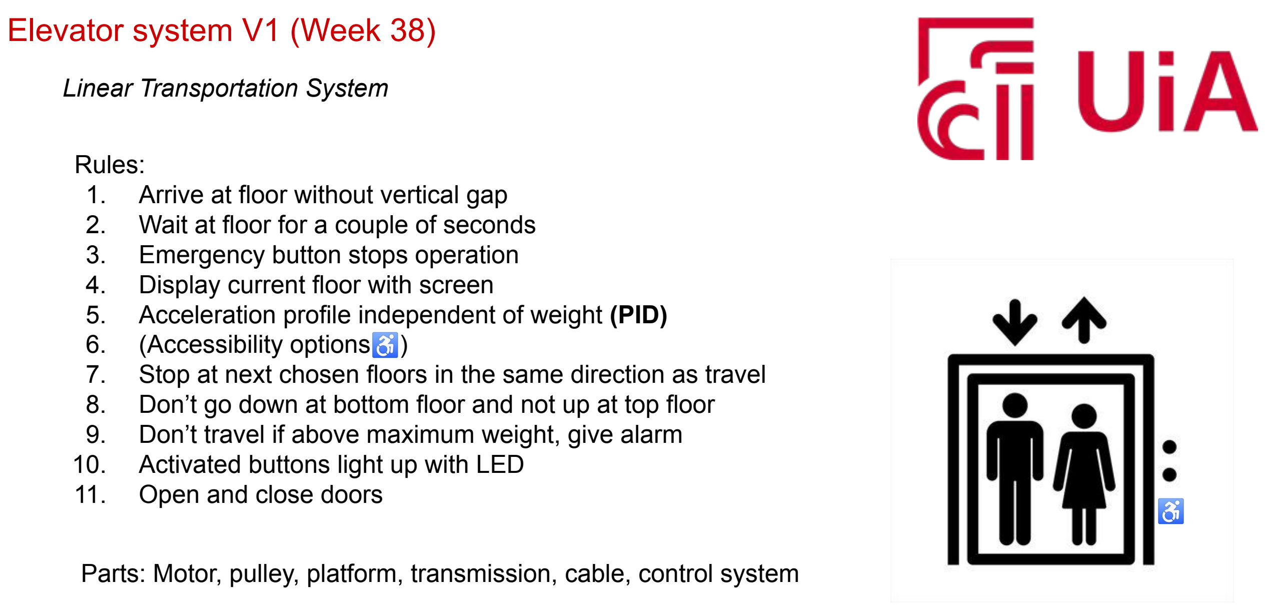
7 Conclusion

The project gave the group a good introduction to how one can structure a system in order to keep it organized. The function of the stepper motor and its different ways to control the steps was explored, where the group reflected the use of each mode. The same applies to the servo motor, where the group learned how to control it using the three modes provided for this driver, and how the encoder can be read. The project also made the group reflect how the speed profile for such a system should look, and how to achieve this, in addition to how a PID-controller may be related to the control algorithm.

A Appendix

A.1 System Requirements

A.1.1 Powerpoint Slide



Elevator system V1 (Week 38)

Linear Transportation System

Rules:
1. Arrive at floor without vertical gap
2. Wait at floor for a couple of seconds
3. Emergency button stops operation
4. Display current floor with screen
5. Acceleration profile independent of weight **(PID)**
6. (Accessibility options♿)
7. Stop at next chosen floors in the same direction as travel
8. Don’t go down at bottom floor and not up at top floor
9. Don’t travel if above maximum weight, give alarm
10. Activated buttons light up with LED
11. Open and close doors

Parts: Motor, pulley, platform, transmission, cable, control system

Figure A.1: Elevator System definition slide

A.1.2 Class Diagram

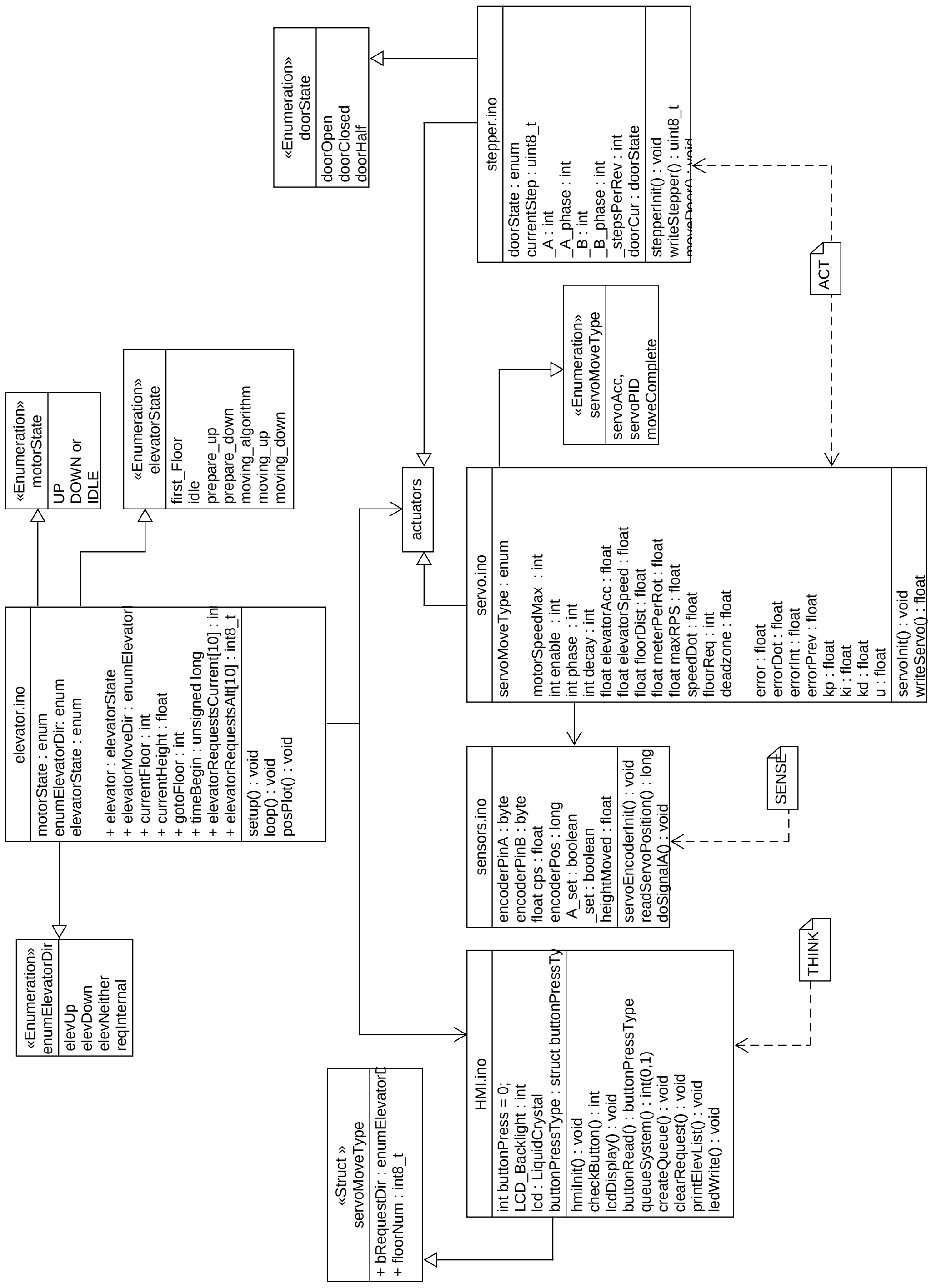


Figure A.2: UML Class Diagram

A.1.3 State Machine Diagram

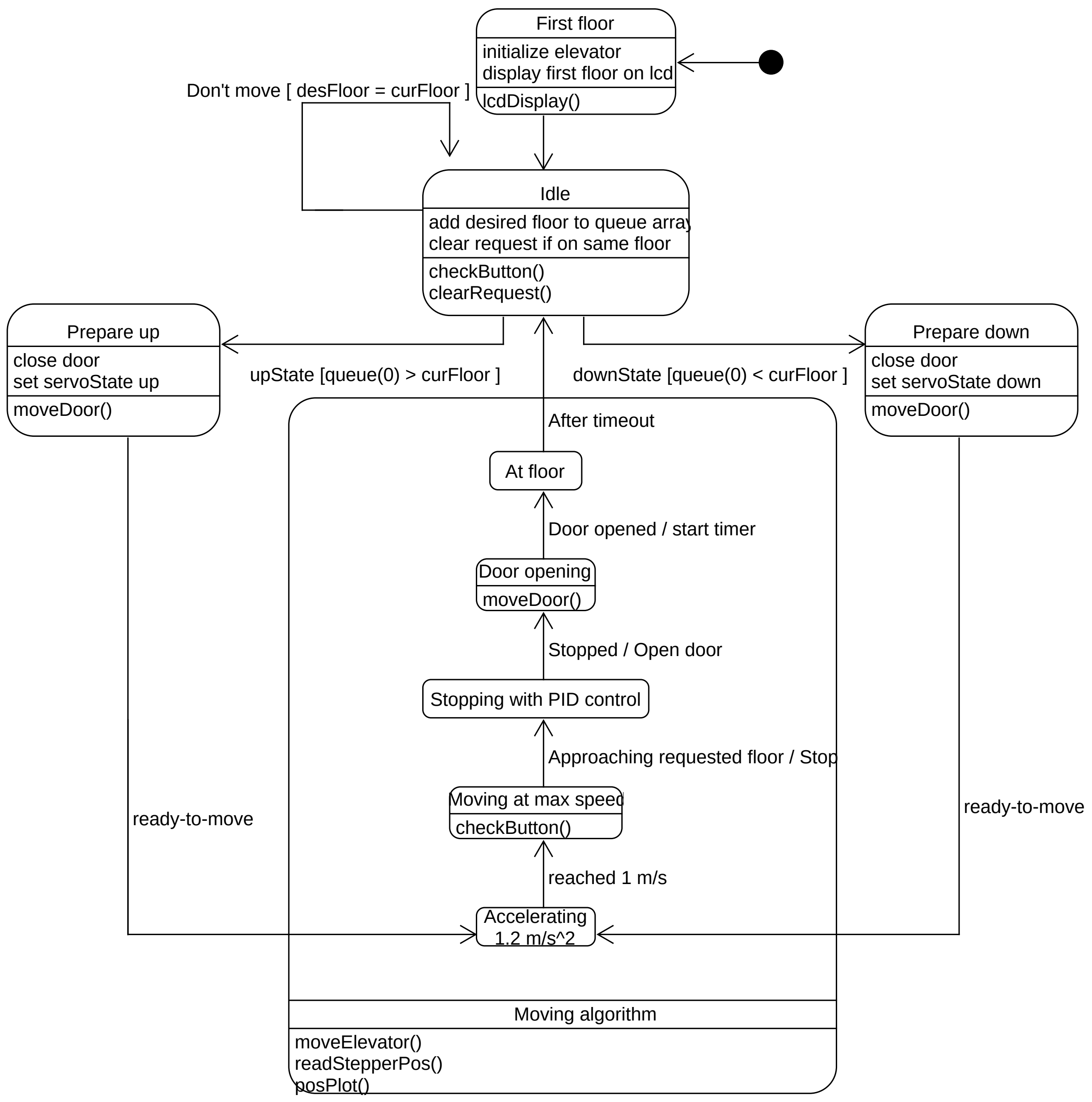


Figure A.3: UML State Machine Diagram

Bibliography

[1] Bjørnar Preus-Olsen Audun Hørthe Helge Nødland. *All in one Servo Lab*. URL: https://uia.instructure.com/courses/13946/files/2251199?module_item_id=507773. (accessed: 7.11.2023).

[2] ORIENTAL MOTOR USA CORP. *Hybrid Stepper Motors and the STEP Hybrid Control Systems*. URL: https://www.orientalmotor.com/stepper-motors/technology/hybrid-stepper-motors-v-hybrid-control.html. (accessed: 20.11.2023).

[3] Thai Son Hoang. *An Elevator System – Requirements Document*. URL: https://uia.instructure.com/courses/13946/files/2251122?module_item_id=507794. (accessed: 21.11.2023).

[4] Lesics. *How does a Stepper Motor work?* URL: https://www.youtube.com/watch?v=eyqwLiowZiU. (accessed: 20.11.2023).

[5] Hipolit Edward Wilczek. *Praktisk anvendelse av DC-/Stepper motoren*. URL: https://uia.instructure.com/courses/13946/files/2341248?module_item_id=532266. (accessed: 1.10.2023).

[6] Hipolit Edward Wilczek. *Praktisk anvendelse av Enkoder*. URL: https://uia.instructure.com/courses/13946/files/2380255?module_item_id=540383. (accessed: 1.10.2023).