# Design Document - Elevator Motor Control Subsystem

## L01\_C\_10 – Bradley Anderson, Connor Wilson, Toby Loveridge, Ahanaf Shahariar Hossain, Justin Tsung

# Table of Contents

Table of Contents

[Design Document - Elevator Motor Control Subsystem 1](#_Toc21900266)

[Table of Contents 1](#_Toc21900268)

[1 - Division of labor and workload acknowledgment 3](#_Toc21900269)

[1.1 – Table of Figures 4](#_Toc21900270)

[2 - Introduction 4](#_Toc21900271)

[2.1 - Scope 4](#_Toc21900272)

[2.2 - Terms, definitions, and abbreviated terms 5](#_Toc21900273)

[3 - Problem definition 5](#_Toc21900274)

[3.1 - Subsystems 6](#_Toc21900275)

[3.2 - Assumptions 6](#_Toc21900276)

[4 - Requirements 7](#_Toc21900277)

[4.1 - Functional requirements 7](#_Toc21900278)

[4.1.1 - MVP Requirements 7](#_Toc21900279)

[4.1.2 - Version 1 Requirements 8](#_Toc21900280)

[4.2 - Performance Requirements 9](#_Toc21900281)

[4.3 - Interface requirements 9](#_Toc21900282)

[4.4 - Interface sign-offs 10](#_Toc21900283)

[4.5 - Constraints 11](#_Toc21900284)

[5 - Design 12](#_Toc21900285)

[5.1 - Conceptual Design 12](#_Toc21900286)

[5.1.1 - Communications Link 13](#_Toc21900287)

[5.1.2 - Priority Queue 13](#_Toc21900288)

[5.1.3 - Sensors 13](#_Toc21900289)

[5.1.4 - Path Finding 13](#_Toc21900290)

[5.1.5 - Motor Control 13](#_Toc21900291)

[5.2 - Design Alternatives 14](#_Toc21900292)

[5.2.1 - Communications Link 14](#_Toc21900293)

[5.2.2 - Sensors 14](#_Toc21900294)

[5.3 - Detailed Design 15](#_Toc21900295)

[5.3.1 - Communications Link 15](#_Toc21900296)

[5.3.2 - Priority Queue 15](#_Toc21900297)

[5.3.3 - Sensors 17](#_Toc21900298)

[5.3.4 - Path Finding 17](#_Toc21900299)

[5.3.5 - Motor Control 20](#_Toc21900300)

[5.4 - Design Traceability 21](#_Toc21900301)

[5.4.1 - Communications Link 21](#_Toc21900302)

[5.4.2 - Priority Queue 22](#_Toc21900303)

[5.4.3 - Sensors 22](#_Toc21900304)

[5.4.4 - Path Finding 22](#_Toc21900305)

[5.4.5 - Motor Control 23](#_Toc21900306)

[6 - References 24](#_Toc21900307)

[7 - Appendix 24](#_Toc21900308)

[7.1 - Appendix A - Motor Controller 24](#_Toc21900309)

[7.2 - Appendix B - Priority Queue 26](#_Toc21900310)

[7.3 - Appendix C - Communications 27](#_Toc21900311)

[7.4 - Appendix D - Sensors 28](#_Toc21900312)

[7.5 - Appendix E - Path Finding 29](#_Toc21900313)

[7.6 - Appendix F - Requirements Document Feedback 31](#_Toc21900314)

# 1 - Division of labor and workload acknowledgment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Member name**  **(first and last name)** | **SID** | **Cohort (ENGG200/300)** | **Sections responsible** | **Overall %** |
| **Connor Wilson** | 44946058 | ENGG300 | * Interface Requirements * Detailed Design - Communications Link * Design Tracability - Communications Link | 20% |
| **Bradley Anderson** | 45229678 | ENGG300 | * Functional & Performance Requirements * Constraints * Conceptual Design * Detailed Design - Priority Queue * Design Tracability - Priority Queue | 25% |
| **Justin Tsung** | 45234531 | ENGG300 | * Interface Requirements * Detailed Design - Motor Control * Design Tracability - Motor Control | 20% |
| **Toby Loveridge** | 45241066 | ENGG200 | * Introduction * Problem Statement * Detailed Design - Path Finding * Design Tracability - Path Finding | 20% |
| **Ahanaf Shahariar Hossain** | 44167822 | ENGG200 | * Conceptual Design * Detailed Design - Sensors * Design Tracability - Sensors | 15% |

## 1.1 – Table of Figures

[Figure 1 - Diagram showing interactions between relevant subsystems and information passed between them 12](file:///C:\Users\Bradley\Downloads\Design+Document+-+Elevator+Motor+Control+Subsystem.doc#_Toc21900191)

[Figure 2 - Reading the data from control box and call the appropriate motorControl function 24](#_Toc21900192)

[Figure 3 - motorControl function 25](#_Toc21900193)

[Figure 4 - updateElevatorQueues() 26](#_Toc21900194)

[Figure 5 - Communications.receive() and Communications.send() 27](#_Toc21900195)

[Figure 6 - Detects which floor the elevator is currently at within the loop. 28](#_Toc21900196)

[Figure 7 - MVP Path Finding implementatio 29](#_Toc21900197)

[Figure 8 - Cases 0 and 1 of the Version 1 implementation 30](#_Toc21900198)

[Figure 9 - Case 3 of the Version 1 implementation 30](#_Toc21900199)

# 2 - Introduction

We have been approached to develop software and communication features of an elevator system, precisely the path finding and motor controls. The path finding is required so that the internal logic of the elevator can decide which floor is the best to travel to first [2], depending on several factors such as elevator state, direction of movement and order of selection. The motor controls will have us focusing on receiving the user or sensor input and translating it for the motor of both the vertical movement and the doors. This will also be where the emergency stop mechanic will be developed.

## 2.1 - Scope

This requirements document will define the functional and non-functional requirements of this elevator project, specifically what our team (L01\_C\_10), who oversee motor control and path planning, will need to complete our section of the project. As stated previously, we are also required to work with other teams closely to achieve transparency between the different parts of the elevator project. This document also has a problem statement, background information and other important information regarding the initialization of the project.

We will be working closely with several other teams focusing on different regions of the elevator system:

* L1\_C20 (Sensors and State Machine)
* L1\_C30 (UI and Comms link)
* L1\_M40 (Wiring of Pins)
* L1\_M50 (Motors and Drive System)
* L1\_M60 (Doors)
* L1\_S80 (Carriage and Counterweights)
* L1\_S90 (Emergency Brakes)

## 2.2 - Terms, definitions, and abbreviated terms

list of terms, definitions, and abbreviated terms

|  |  |
| --- | --- |
| **Abbreviation** | **Definition** |
| SRS | System Requirements Specification |
| MVP | Minimum Viable Product |
| m/s | Meters per Second |
| API | Application Programming Interface |
| RPM | Revolutions Per Minute |
| PWM | Pulse Width Modulation |
| REQ | Requirement (Version 1) |
| PRQ | Performance Requirement |
| FIRQ | Functional Interface Requirement |

# 3 - Problem definition

The customer has requested an elevator system to be developed, that requires a variety of features. It will allow for easy traversal between floors in a multi-storey building, consisting of a carriage where 2-5 kilos of weight can be supported. There will be motors, counterweights and a pulley system to allow for movement. Doors will respond to user inputs for open and close, and user controls will allow for movement between floors. The elevator will also have safety precautions such as guard rails, auto sensing and brake system.

To complete this project, the engineers will focus on specific subsections of the overall system. Our group will focus on developing controls for the motor, and the internal path-finding logic. Our problem is implementing code to control the motor in ways dictated by user input, as well as developing an algorithm that decides in which order the elevator should make stops and move up and down.

## 3.1 - Subsystems

There are 3 main subsections of groups: the Communications teams, the Motion teams and the Structural teams. our team L\_C\_01 is a Comms team, so we will be focusing on the subsystem of communication and control of the automated system as a whole, while the motions teams focus on the motorized aspects of the system, and the structural teams focus on the overall building structure of the elevator system.

We will be working closely with other Comms teams C\_20 and C\_30. C\_20 are focusing on the sensor programming and the state machine for the elevator, and C\_30 are focusing on the user interface and communications link.

## 3.2 - Assumptions

|  |  |
| --- | --- |
| **Assumption** | **Dependent Requirement** |
| Assumption 1: It is assumed that a UI will be in place for users to choose whether to send the elevator up or down. | * MVP03 * REQ03 * PRQ01 |
| Assumption 2: It is assumed that the elevator will be moved by a motor up and down, reaching a max speed and slowing upon floor arrival. | * MVP04 * REQ01 * REQ02 * PRQ3 * PRQ4 * FIR\_16 * FIR\_17 * FIR\_21 |
| Assumption 3: It is assumed an emergency brake system will be in place, with user input. | * MVP02 * REQ06 |
| Assumption 4: Sensors will be able to detect the state of the elevator (i.e. Floor Number) and inform the path finding system and motor controls. | * MVP03 * MVP04 * REQ03 * REQ04 * REQ05 * FIR\_11 * FIR\_17 * FIR\_21 |
| Assumption 5: The doors will be opened and closed once the button is pressed, controlled via a servo. | * MVP01 * REQ02 * REQ03 * PRQ2 * FIR\_16 |
| Assumption 6: It is assumed that the elevator will Start from floor 1 from the powered off state. | * MVP04 * REQ4 * REQ05 |

# 4 - Requirements

## 4.1 - Functional requirements

All things that are required of the motor controller are the operation of the carriage motor located at the top of the elevator shaft as well the door motor system. In addition, the motor controller has to properly plan out a path of floors for the elevator carriage to visit when buttons are pressed.

### ****4.1.1 - MVP Requirements****

These are the requirements of the MVP that is to be completed by Week 7, this product will be built upon to create the product that will be demonstrated later in Week 13. These are as follows:

**MVP01** - The system shall be able to open and close the doors on the elevator.

**MVP02** - The system shall be able to stop the elevator in the event of an emergency. This will occur in the event that the emergency stop button is pressed or the elevator loses power.

**MVP03** - The system shall be able to correctly receive input from the controller. This will include going up and down a single floor, opening or closing the doors and initiating the emergency stop.

**MVP04** - The system shall be able to correctly ascend and descend the elevator to the floor directly above or below it. When the up or down button is pressed the elevator moves accordingly.

### ****4.1.2 - Version 1 Requirements****

This first version of the product is the one that will use the MVP as a basis to improve upon and add more functionality and refine the already existing features present in the MVP. This version of the product will be demonstrated in Week 13 and has the following requirements:

**REQ01** - The system shall be able to move the elevator upwards and downwards at a smooth and constant rate, irrespective of how much weight is inside the elevator carriage. This will be done to keep the movement of the elevator consistent, regardless of its weight.

**REQ02** - The system shall be able to open and close the doors when the appropriate button has been pressed. In addition, if the doors are open and the order to go to a different floor is received, the doors will close automatically before moving to the requested floor.

**REQ03** - The system shall be able to take input from the elevator’s control panel and perform the correct action. This will improve upon **MVP03** by replacing ‘up’ and ‘down’ with numbered floors from 1 - 5. This will be done by interpreting byte string inputs from the Communications team and storing them in a queue to be read from.

**REQ04** - The system shall be able to keep track of what floor the elevator is currently on and if it’s ascending or descending.

**REQ05** - The system shall be able to prioritize visiting floors correctly. For example, if the elevator is on the 2nd floor and the 4th, 5th and 1st floor buttons are pressed, the elevator will visit the closest floor in the upward direction that has been queued. It will then visit all the queued-up floors above that before going to the closest queued floor in the downward direction.

**REQ06** - The system shall be able to stop the elevator in the event of an emergency. This will occur when the emergency stop button is pressed or the elevator loses power.

**REQ07** - The system shall be able to appropriately handle errors and send error messages to the UI System. Such errors will include if the message received from the Communications Link is invalid.

**REQ08** - The system shall be able to keep track of the speed that the elevator is moving.

## 4.2 - Performance Requirements

**PRQ1** - The motor system shall begin driving the elevator within 3 seconds once the doors are closed. This will be done to avoid holdups of the elevator and keep the system running smoothly.

**PRQ2** - The elevator shall move at a speed equal to that of a standard elevator scaled down to fit the scale model, the standard speed being 10 - 20m/s. This will be done to ensure the lift moves at a reasonable speed for the scale of the model that the elevator will be situated in.

**PRQ3** - The elevator shall start off slow and speed up as well as slowing down as it approaches a floor before coming to a stop. This will be done to avoid sudden starts and stops.

## 4.3 - Interface requirements

The following interface requirements detail things that are required from other subsystems / are required from us for the elevator system as a whole to operate efficiently. This list of requirements has been taken from the L1 Interface Requirements document in the L1 shared space.

**FIRQ\_8 -** The Motor Controller subsystem shall receive an Integer “weight” specifying the weight of the load inside the elevator car from the Sensor subsystem.

**FIRQ\_11 -** The Motor Controller subsystem shall receive an Integer (dist) specifying the distance traveled between levels from the Sensors subsystem.

**FIRQ\_15 -** The Motor Control subsystem shall receive a bit string that represents relative commands from the communication UI team.

* “E”: represents a Boolean value for the emergency stop
* “U”: represents up, “D” represents down
* “O”: represents open “C”: represents Close
* “0-1”: represents bit parity

**FIRQ\_16** **-** The Motor Control subsystem shall be able to send a PWM signal to the motor.

**FIRQ\_17 -** While in the moving state the Motor Controller subsystem shall receive an integer representing the level from the Sensor subsystem.

**FIRQ\_21 -** From the Motor control subsystem, the User Interface subsystem shall receive the following information in addition to the user inputs for MVP:

* Lift going up/down (string U/D)
* Lift has stopped (String S)
* Emergency stop pressed (String E)
* For NMVP:
  + Doors closing (String C)
  + Doors opening (String O)
  + Error information (String Er)

**FIRQ\_22 -** The Motor Control subsystem shall send a confirmation message to the Control Box subsystem detailing that the two micro controllers (Arduino Mega and Uno) are connected.

## 4.4 - Interface sign-offs

[Julian Cesaro](https://fse-engg.atlassian.net/wiki/people/5d3f976f4ab9030da141ee8b?ref=confluence) - L01\_C\_20 Has signed off on this requirement - **FIRQ\_8.**

[Bradley Anderson](https://fse-engg.atlassian.net/wiki/people/5b6240d2cee8472cde7b37b7?ref=confluence) - L01\_C\_10 Has signed off on this requirement - **FIRQ\_8.**

[Julian Cesaro](https://fse-engg.atlassian.net/wiki/people/5d3f976f4ab9030da141ee8b?ref=confluence) - L01\_C\_20 Has signed off on this requirement - **FIRQ\_11.**

[Bradley Anderson](https://fse-engg.atlassian.net/wiki/people/5b6240d2cee8472cde7b37b7?ref=confluence) - L01\_C\_10 Has signed off on this requirement - **FIRQ\_11.**

[YEFAN LI](https://fse-engg.atlassian.net/wiki/people/5b624422ffd0a32d7917bbd5?ref=confluence) - L01\_C\_30 has signed off on this requirement **FIRQ\_15.**

[Bradley Anderson](https://fse-engg.atlassian.net/wiki/people/5b6240d2cee8472cde7b37b7?ref=confluence) - L01\_C\_10 Has signed off on this requirement **FIRQ\_15.**

[Peter Sotirios](https://fse-engg.atlassian.net/wiki/people/5d484c6aa23f060c10836ca4?ref=confluence) - L01\_M\_40 Has signed off on this requirement - **FIRQ\_16.**

[Bradley Anderson](https://fse-engg.atlassian.net/wiki/people/5b6240d2cee8472cde7b37b7?ref=confluence) - L01\_C\_10 Has signed off on this requirement - **FIRQ\_16.**

[Julian Cesaro](https://fse-engg.atlassian.net/wiki/people/5d3f976f4ab9030da141ee8b?ref=confluence) - L01\_C\_20 Has signed off on this requirement - **FIRQ\_17.**

[Bradley Anderson](https://fse-engg.atlassian.net/wiki/people/5b6240d2cee8472cde7b37b7?ref=confluence) - L01\_C\_10 Has signed off on this requirement - **FIRQ\_17.**

[YEFAN LI](https://fse-engg.atlassian.net/wiki/people/5b624422ffd0a32d7917bbd5?ref=confluence) - L03\_C\_30 has signed off for this requirement - **FIRQ\_21.**

[CONNOR WILSON](https://fse-engg.atlassian.net/wiki/people/5b62662335712e3b81c69e8c?ref=confluence) - L01\_C\_10 has signed off for requirement - **FIRQ\_21.**

[YEFAN LI](https://fse-engg.atlassian.net/wiki/people/5b624422ffd0a32d7917bbd5?ref=confluence) - L01\_C\_30 has signed off on this requirement - **FIRQ\_22.**

[CONNOR WILSON](https://fse-engg.atlassian.net/wiki/people/5b62662335712e3b81c69e8c?ref=confluence) - L01\_C\_10 has signed off for requirement - **FIRQ\_22.**

## 4.5 - Constraints

Constraints for the motor controller subsystem include:

* 256 KB of flash memory on the Arduino Mega to store and run any code written. This will limit the amount of code that can be used to achieve the solution and therefore data structures and processes should aim to be as small as possible
* An MVP is expected by Week 7, and the finished product by Week 10 for testing. Giving an 8-week time frame to develop a fully finished product. This can be mitigated by having clear deadlines and keeping track of progress on tasks for the project.
* Our team consists of 5 people, responsibilities need to be divided between communication with other teams, programming and documentation.
* The team has been given one Mega as a loan, so testing will need to be done in a group. Testing can be done during workshop times as well as during allocated meeting times with other teams to ensure that each team member and other teams are up to date with what’s going on in the project.
* The programming environment is the Arduino API which uses C++, this also limits us to the use of libraries and structures compatible with the Arduino IDE [1].
* Any electrical components need to be compatible with said Arduino API. A list of electrical and mechanical components we’re allowed to use has been provided for us, all of which are compatible with the Arduino API.

# 5 - Design

## C:\a7ef88f550627cf99b58776daf2634a35.1 - Conceptual Design

Figure 1 - Diagram showing interactions between relevant subsystems and information passed between them

The motor controller system has 3 subsystems within it: Priority Queue, Path Finding and Motor Control. This system relies on 2 other systems managed by different groups these being the Communications Link/UI being managed by L1\_C\_30 and the Sensor and State Machine being managed by L1\_C\_20.

The roles of each of these systems and their interface interactions are as follows:

### ****5.1.1 - Communications Link****

The Communications Link is responsible for sending orders of what the elevator should do (Moving between floors, opening and closing the doors and the emergency stop signal). It will send its commands via a bit string to the Priority Queue subsystem which keeps track of all commands that were given to it. This system will also receive updates from the Motor Control subsystem as to if the elevator has arrived at its destination as well any errors that may have occurred during run time.

### ****5.1.2 - Priority Queue****

The Priority Queue holds the commands given to it by the Communications Link. Commands that come through will be placed in a priority queue so that orders given to it can be executed in order. Commands that are popped off the queue are then sent to Path Finding which determines how the elevator will respond to the given command. The only exception to this is when the emergency stop bit string is sent. In this case the priority queue skips the path finding and immediately tells the motor controller to halt all motors.

### ****5.1.3 - Sensors****

The Path Finding system will take commands from the Priority Queue. Once it receives a command it will then need to receive information from the Sensors to determine which floor it’s on. Based on this info the Path Finding system will plan the path the elevator needs to take. From here Path Finding will instruct Motor Control on how to run the motors to complete the command popped off the queue. Commands include characters such as “O” or “C” or opening/closing the doors, the numbers “1” - “5” representing a floor to visit or an “E” representing the emergency stop.

### ****5.1.4 - Path Finding****

The Sensors system will oversee various sensors and switches responsible for keeping track of what floor the elevator has reached and the weight that’s inside the elevator carriage. The sensors will constantly send information to Path Finding so that it can plan out the route the elevator will end up taking on a floor-by-floor basis.

### ****5.1.5 - Motor Control****

The Motor Control is responsible for driving the motors that will raise and lower the elevator carriage as well as the motor responsible for opening/closing the doors. It will also keep track of the direction the elevator is moving and will send status messages to the Communications Link when the elevator arrives at its destination. The Motor Control will receive instructions from Path Finding as to which way the motors need to be driven to reach a certain floor or to open or close the doors. The only interaction between the Priority Queue and Motor Control is if the Priority Queue receives an emergency stop command, which will override all currently queued commands and instantly initiate the emergency stop procedure.

## 5.2 - Design Alternatives

### ****5.2.1 - Communications Link****

There were several considerations for how and in what form inputs would be passed to the Priority Queue and Path Finding.

There are two forms of being able to send data to the system, via Bluetooth link and via a wired connection directly from the Arduino Uno. The wireless Bluetooth approach was reserved for a non-MVP system; however, this requires no additional implementation for the Motor Controller, Path Finding or Priority Queue.

The form of the input had a few different considerations. Input and output were planned to be sent and received as either integers, chars or strings. Integers allowed for easy placement inside of queues but makes it harder to receive commands such as “UP” or “DOWN”.

Chars solve this problem as both single digit numbers and letters can be received, so “UP” and “DOWN” become “U” and “D”, but conversion from chars to integers to put into the Priority Queue is difficult to do.

Strings solve the issue of char to integer conversion as a String has the toInt() function, which is used to place inputs such as “1” or “2” into the Priority Queue, while still keeping inputs like “U” or “D” as strings.

### ****5.2.2 - Sensors****

There are sensors that keep track of various aspects of the lift and allow the system to know when certain conditions are met, such as what floor the carriage is on and if the doors are open or closed.

Originally, limit switches were planned for use on every floor as a method of detecting when the lift arrived at that floor. The lift would hit the switch, triggering a signal change, indicating that the lift is at that floor. This idea remains but is only used for floors 1 and 5.

The use of Hall-effect sensors was also changed, originally these were used for the doors to detect if they were closed or not. This design was dropped for the doors, so the sensors were re-purposed for the lift floor tracking. Now Hall-effect sensors are used on floors 2 - 4, with the proximity of the elevator to that floor being a threshold as to whether a LOW or HIGH signal is sent from the sensor.

The changing sensor just changes the implementation of the lift carriage tracking, rather than expecting a switch to be triggered on floors 2 - 4, a magnetic threshold needs to be exceeded instead. This has the potential of enabling false positives on those floors. This can be addressed with the PWM encoder on the motor, if the value for that isn’t within a reasonable range of the floor if the signal changes then it’s most likely the sensor was falsely triggered and should be ignored.

## 5.3 - Detailed Design

### ****5.3.1 - Communications Link****

The communications link is responsible for transmitting orders to tell the elevator what action to perform (moving between floors and the emergency stop signal). It will send its commands via a bit string to the data variable. If data is equal to “E” then we will immediately call our emergency stop function which will force the elevator to a halt. Otherwise we will place the “data” variable into the priority queue which will explained below.

The process of receiving information is done via a function called Communications.recieive(). This function will take any information received from the Serial 2 port and turn it into a readable string. The system is always listening for input on this port, just in case an emergency signal needs to be sent to the Motor Controller to trigger the emergency stop. (Function can be seen in Appendix C, Figure 5)

We also have access to a non-MVP specification to send information back to the control box; this function is called “Communications.send()”. This function will simply take the given String and write it to the serial 2 port which will be sent back to the control box team. (Function can be seen in Appendix C, Figure 5)

### ****5.3.2 - Priority Queue****

This subsystem utilises a few global variables and functions, these include:

* **queueAscending()** - A function used as a comparitor to sort a queue of integers in ascending order.
* **queueDescending()** - Another comparitor that sorts a queue of integers in descending order.
* **elevatorStatus** - An integer representing in which direction the lift is travelling. 1 for ascending, -1 for descending, 0 for stationary.
* **currFloor** - An integer representing the last floor that the elevator reached. Is updated by the Sensors subsystem whenever the elevator reaches a floor to be that floor.
* **queueFloorsAbove** - A PriorityQueue that uses the queueAscending function to store all floors that are above the elevator carriage in ascending order to be visited.
* **queueFloorsBelow** - A PriorityQueue that uses the queueDescending function to store all floors below the elevator carriage in descending order to be visited.
* **inQueue** - An array of Booleans of size 5 representing whether a floor has been added to either queueFloorsBelow or queueFloorsAbove. For example if inQueue[2] is set to true, then floor 3 exists in either of the two queues already and doesn’t need to be added again. This avoids duplicates of floors being added to the same queue or potentially both queues.
* **updateElevatorQueues()** - The main function as part of this system. Takes an input itemToAdd which is a string representing the numbers “1” - “5”. This will be converted to an int for use in the function.

Both queueFloorsBelow and queueFloorsAbove utilize an imported PriorityQueue library [3] which creates queues that use comparitor functions to sort their contents, just like a PriorityQueue in Java. The system is split into two separate queues, one for ascending and one for descending which Path Finding uses to determine what floors to visit when travelling either up or down the elevator shaft.

The idea behind the design is that Path Finding will choose the first queue that has an item placed into it when the lift is stationary. The lift will then visit any additional floors placed on the queue that it’s currently looking at. Once the queue it’s looking at is empty, Path Finding will then switch over to the other queue to visit any floors placed on the queue. This cycle continues until the elevator has no floors it needs to visit in either queue, so it will wait in its current location.

This design also allows for the dynamic placement of new floors as the lift is moving. For instance, if floor 5 is the currently queued floor above the lift currently at floor 2, you can push the button for floor 4 and the lift will now change its next floor to visit to floor 4, before visiting floor 5.

When the Communications Link sends an input that denotes a floor to visit, the function updateElevatorQueues() is called on the received input, the code for which can been seen in Figure 4 of Appendix B.

When the function is called, the input itemToAdd is checked to see if it’s a valid input representing a floor to visit. If so then it’s converted to an int floorToAdd. After this, it’s index in inQueue is checked (as stated before the respective index is floorToAdd - 1). If this check results in a false, then the function continues on as the input doesn’t yet exist in either queueFloorsAbove or queueFloorsBelow and can be safely added.

At this point, the floor that’s being added is compared to the floor that the elevator carriage last reached, if floorToAdd is greater than currFloor, it’s above where the elevator is currently located so it’s added to queueFloorsAbove and the floor’s index in inQueue is set to true. This indicates that the floor is now located in one of the two queues and shouldn’t be added again until removed.

If floorToAdd < currFloor then the floor being added is below where the elevator is, so it gets added to queueFloorsBelow and the corresponding index in inQueue is set to true.

In the case where floorToAdd = currFloor that means the elevator has moved off from that floor but is now travelling to a different floor. In this case, elevatorStatus is checked to see if the elevator is ascending or descending. If elevatorStats = 1 when floorToAdd = currFloor, then the elevator has just left currFloor in the upwards direction, so floorToAdd will be placed in queueFloorsBelow. If elevatorStatus = -1, then floorToAdd is placed into queueFloorsAbove.

If currFloor is equal to 5 or 1, then there’s no more floors above or below the lift so floorToAdd is added to queueFloorsBelow or queueFloorsAbove respectively

The result of this function is that both either queueFloorsAbove or queueFloorsBelow will be updated to have floorToAdd in the correct place depending on which queue it was put in. The floor will then be visited when it comes to the front of its respective queue when Path Finding pops it off.

### ****5.3.3 - Sensors****

The Sensors system will be in charge of various sensors and switches responsible for keeping track of what floor the elevator has reached and the weight that’s inside the elevator carriage. The sensors will constantly send information to Path Finding so that it can plan out the route the elevator will end up taking on a floor-by-floor basis.

The sensors system uses these global variables:

* **level\_1\_state, level\_5\_state** - The signal received from the limit switch on the 1st and 5th floors respectively, will either be HIGH or LOW.
* **level\_2\_state, level\_3\_state, level\_4\_state** - The signal received from the Hall-effect sensors on the 2nd, 3rd and 4th floors. The signal of HIGH or LOW changes dependent on a threshold when the elevator gets close enough to the sensor to trigger a signal change.
* **currFloor** - A global variable described before in the Priority Queue section. This subsystem will update this value whenever any one of the level states is triggered, indicating the elevator is at that floor.

The code seen in Appendix D, Figure 6 is run in the main loop, constantly looking for signal changes on any floor. This loop doesn’t stop until the elevator system is switched off.

### ****5.3.4 - Path Finding****

Path Finding has 2 separate implementations for the MVP and Version 1 product, both of which will be covered here.

Path Finding’s MVP solution uses these variables:

* **data** - A string that’s constantly updated based on input received from the Communications Link, is used to keep track of commands given by the UI system about what floor to go to and if an emergency stop is required.
* **data2** - A string that is based on data when the elevator needs to move. The current value of data is captured so that the Path Finder can keep track of the command that was last given by the UI. This way, data can keep updating without affecting the current command the lift is focused on.
* **prevFloor** - A copy of currFloor before the lift starts moving. This will be used to compare against currFloor to see if the lift has successfully made it to a different floor.
* **currFloor** - Updated by the sensors when the lift reaches a new floor, this is compared against prevFloor while the lift is moving. The moment currFloor becomes different to prevFloor, the lift has made it to a different floor.

The code for the MVP implementation of the path finding can be seen in Appendix E, Figure 7.

When data is updated when the UI give the lift commands to perform, data2 has data’s contents copied into it and prevFloor has currFloor’s contents copied to it. Once this happens data2 is checked to see what command is given to it. This has one of 4 outcomes:

1. If data2 = ““, there’s no command to execute, so the elevator stays in place by calling elevatorMovement(“S”) which stops the elevator.
2. If data2 = “U”, the elevator needs to head up 1 floor. At this point elevatorMovement(“U”) is called and the elevator will move up. at the point where currFloor > prevFloor, the elevator has moved up at least one floor, so elevatorMovement(“S”) is called and data2 will be set to the empty string ““.
3. If data2 = “D“, the elevator needs to head down 1 floor, this calls elevatorMovement(“D”) to move downwards. When currFloor < prevFloor then the elevator has moved down at least one floor, so the elevator is stopped and data2 is set to ““.
4. If data = “E”, the elevator needs to emergency stop. This calls elevatorMovement(“E”) which stops the elevator but keeps the motor on so that it can hold itself with the resistance from the motor. During any of the other cases, data is still being updated and in the event of data = “E”, it’s required to skip copying to data2 and just immediately stop the elevator.

Path Finding’s version 1 implementation uses these variables:

* **queueCase** - An integer that indicates the order in which to use queueFloorsAbove or queueFloorsBelow. There are 5 different values.

1. 0 is for when both queueFloorsAbove and queueFloorsBelow are empty
2. 1 is for when queueFloorsBelow is the only queue with items in it.
3. 2 is for when queueFloorsAbove is the only queue with items in it.
4. 3 is for when queueFloorsAbove is filled before queueFloorsBelow is filled.
5. 4 is for when queueFloorsBelow is filled before queueFloorsAbove is filled.

* **queueFloorsAbove** - A priority queue containing all floors above the current floor that have been chosen to go to. The floor at the head of the queue is the next floor in the upward direction that’s closest to the lift.
* **queueFloorsBelow** - A priority queue containing all floors below the current floor that have been chosen to go to. The floor at the head of the queue is the floor in the downwards direction that’s closest to the lift.
* **elevatorStatus** - This is an integer that represents which direction the lift is travelling in. 1 for ascending, 0 for stationary, -1 for descending. This is used by the priority queues to determine what floors are above/below the lift, so it’s required to change as the lift is instructed to ascend or descend.
* **currFloor** - As stated in the MVP section this is the floor that the elevator has last reached. When the elevator reaches a floor, this value is updated to reflect the floor the elevator has just reached.

Both queueFloorsAbove and queueFloorsBelow are checked to see if they have items inside of them. Depending on their contents, queueCase is updated accordingly.

For instance, filling queueFloorsAbove with a floor will set queueCase to 2, since it’s the only queue with items in it. However, adding a floor to queueFloorsBelow will change queueCase to 3 since queueFloorsAbove was filled prior to queueFloorsBelow getting a floor added to it.

The path finding this time around works with a series of cases, case 0 just has the elevator stationary with elevatorStatus set to 0 since both queues are empty, so there’s no floors to visit.

The code for cases 0, 1 and 3 can be seen in Appendix E, Figures 8 and 9 respectively.

Cases 1 - 4 deal with when there’s items in any combination of the two queues. In the example of case 1, while there’s still items in queueFloorsBelow, elevatorStatus is set to -1 and the elevator will be moving down the shaft.

The lift is constantly checking if the head of the queue is equal to currFloor. If it is, then the current head of queueFloorsBelow is popped off and the elevator stops at that floor for 3 seconds, giving it a moment where the elevator is stopped at that floor.

This continues until queueFloorsBelow is empty, at which point the lift will stop moving and queueCase is set to 0, since now there’s no items in either queue.

Case 2 handles only if queueFloorsAbove is the only queue with items in it and operates the same as Case 1, just with the elevator ascending instead of descending and elevatorStatus set to 1 instead of -1.

As can be seen in Appendix E, Figure 9, Case 3 is a case where queueFloorsAbove gets filled first then queueFloorsBelow gets floors added to it after the lift has already started using the ascending queue. In this instance the elevator runs just like it would in case 1 or 2, visiting every floor in the queue until the queue is empty.

At this point however, rather than stopping the lift and setting queueCase to 0, queueCase will be set to 1 in this instance, since if queueFloorsAbove has been emptied and queueFloorsBelow has items in it, then it’s exactly the same as Case 1, so that’s what needs to be run next.

### ****5.3.5 - Motor Control****

The Motor Control is responsible for driving the motors that will raise and lower the elevator carriage as well as the motor responsible for opening/closing the doors. It will also keep track of the direction the elevator is moving and will send status messages to the Communications Link when the elevator arrives at its destination. The Motor Control will receive instructions from the Path Finding system as to which way the motors need to be driven to reach a certain floor or to open or close the doors. The only interaction between the Priority Queue and Motor Control is if the Priority Queue receives an emergency stop command, which will override all currently queued commands and instantly initiate the emergency stop procedure.

The first motor control function takes the input from the control box i.e. a string of “U” up, “D” down, “E” emergency stop. This can be seen in Appendix A, Figure 2. Once the corresponding string is received by the communications, the motorControl function receives which state the motors should be in. As seen in Appendix A Figure 3, each state is parsed to a case. There are 4 cases for the movement of the elevator:

* **Case 0** - The motor should stop.
* **Case 1** - The motor should go in reverse (the elevator carriage will move down)
* **Case 2** - The motor should stop and the power to the motor pin is turned off (emergency stop).
* **Case 3** - The motor should go forwards (the elevator carriage will upwards down)

For each of the different cases. digitalWrite and writeMicroseconds are used to control the motors. When digitalWrite is on “LOW” it means that the motor pin is enabled whereas “HIGH” the pin is disabled. To move the motor forwards or reversed is depended on the value parsed into writeMicroseconds.

## 5.4 - Design Traceability

### ****5.4.1 - Communications Link****

* **MVP02:** The system shall be able to stop the elevator in the event of an emergency. - This will occur in the event that the emergency stop button is pressed or the elevator loses power. For the emergency stop to be read it must first be received through the communication node.
* **MVP03:** The system shall be able to correctly receive input from the controller. This will include going up and down a single floor, opening or closing the doors and initiating the emergency stop. - When we call commuinications.receive() we will read in any data sent to us, which in turn will be processed by the queue.
* **MVP04:** The system shall be able to correctly ascend and descend the elevator to the floor directly above or below it. - When the up or down button is pressed the elevator moves accordingly. When we call commuinications.receive() we will read in any data sent to us, which in turn will be processed by the queue.
* **REQ02:** The system shall be able to open and close the doors when the appropriate button has been pressed. - When we call commuinications.receive() we will read in any data sent to us, which in turn will be processed by the queue.
* **REQ03:** The system shall be able to take input from the elevator’s control panel and perform the correct action. - When we call commuinications.receive() we will read in any data sent to us, which in turn will be processed by the queue.
* **REQ06:** The system shall be able to stop the elevator in the event of an emergency. This will occur in the event that the emergency stop button is pressed or the elevator loses power. - When we call commuinications.receive() we will read in any data sent to us, As it will be the emergency stop, it will automatically be dealt with straight away.
* **FIRQ\_15:** The Motor Control subsystem shall receive a bit string that represents relative commands from the communication UI team.
  + “E”: represents a Boolean value for the emergency stop
  + “U”: represents up, “D” represents Down
  + “O”: represents open “C”: represents Close
  + “0-1”: represents bit parity

When we call commuinications.receive() we will read in any data sent to us. It will be in the form shown above.

* **FIRQ\_21:** From the Motor control subsystem, the User interface subsystem shall receive the following information in addition to the user inputs for MVP:
  + Lift going up/down (string U/D)
  + Lift has stopped (String S)
  + Emergency stop pressed (String E)
  + For NMVP:
    - Doors closing (String C)
    - Doors opening (String O)
    - Error information (String Er)
* When we call commuinications.send() we will send them data based on the string stated above.

### ****5.4.2 - Priority Queue****

* **REQ3:** The system shall be able to take input from the elevator’s control panel and perform the correct action. - Anytime the system reads an input corresponding to a floor the queuing function is called. Provided the input is a valid floor, that floor will be added to one of the queues (Unless already present on one of the two queues) which will be visited at some point when Path Finding reaches that floor in the queue.
* **REQ5:** The system shall be able to prioritise visiting floors correctly. - The priority queues allow for floors above and below the lift to be correctly organised in the order to be visited in. Items are popped off the queue as they’re visited and the current queue is always emptied before changing to the other queue, maintaining this cycle as long as floors are queued up much like a regular lift. Floors can also be added while the elevator is in transit. For example, if the descending queue has [3, 1], then adding floor 4 onto the descending queue while the lift is coming down will result in a queue of [4, 3, 1] with floor 4 being the new first floor to stop at.

### ****5.4.3 - Sensors****

* **REQ04:** The system shall be able to keep track of what floor the elevator is currently on and if it’s ascending or descending. - This is where the sensor loop steps in. The sensors will figure out which floor the elevator is on at that moment. The variable currFloor will provide this information to the motor control. Unless this information is passed to the motor control, the elevator will not know if it should go up or down. I.e. if the elevator is at level 2 and the user presses floor 1, the sensors will let the motor control know that the lift is at floor 2 and the lift must go down.

### ****5.4.4 - Path Finding****

Requirements met directly by this implementation of the path finding:

* **REQ03:** The system shall be able to take input from the elevator’s control panel and perform the correct action. - The path finding system correctly takes in a floor selection from the control panel, adds it to the correct queue and will eventually take you to that selected floor in accordance with the algorithm (i.e. the elevator is going to floor 2 from floor 4, floor 1 is selected, the queue will then contain [2, 1]. The peak is taken (2) and is given to the motor control to take the elevator there).
* **REQ04:** This is accomplished within the variable currFloor in the priority queue, which is a global variable that is updated each time the sensor detects a floor change. This is used for finding the correct path.
* **REQ05:** The system has been shown to correctly prioritise an array of different floors into a path for the elevator to follow. This has not been tested with live inputs, but the algorithm correctly organises the floors into their correct queue, in the correct position. The test conducted also accounted for a variety of situations such as multiples of the same floor selected, random floor selection and timing issues (e.g. floor picked at the same time a floor is reached).
* **REQ06:** When emergency brakes are applied, they will need to also disregard the path the elevator is currently taking. This will be done outside of the path finding code, as the motor control will be updated first on the emergency to allow for a quick response in a possibly critical situation.

### ****5.4.5 - Motor Control****

Requirements met directly by the implementation of motor control:

* **MVP01:** The system shall be able to open and close the doors on the elevator. - The motor control code function incorporates the door function to open and close the doors.
* **MVP02:** The system shall be able to stop the elevator in the event of an emergency. This will occur in the event that the emergency stop button is pressed or the elevator loses power. - Once the String “E” is received by the communications, the motors will disable the motor pin and setting the speed of the elevator to stop position.
* **MVP04:** The system shall be able to correctly ascend and descend the elevator to the floor directly above or below it. When the up or down button is pressed the elevator moves accordingly. - The motor function can move the elevator ascending or descending by switching the speed of the elevator. This is done by rotating the motor clockwise or anti-clockwise.
* **PRQ2:** The elevator shall move at a speed equal to that of a standard elevator scaled down to fit the scale model, the standard speed being 10 - 20m/s. - The motor function sets a standard speed which the elevator will move at a constant speed.
* **PRQ3:** The elevator shall start off slow and speed up as well as slowing down as it approaches a floor before coming to a stop. - The motor function uses the sensor information as well as the encoder to determine when to slow down or speed up.
* **REQ08:** The system shall be able to keep track of the speed that the elevator is moving. - The motor function calculates the speed of the elevator by determining the number of revolutions the motor has done in a second.

# 6 - References

[1] Arduino Language Reference Page, <https://www.arduino.cc/reference/en/>, Arduino 2019.

[2] Disk Scheduling/Elevator Algorithm, <https://web.archive.org/web/20080606005055/http://www.dcs.ed.ac.uk/home/stg/pub/D/disk.html>, The University of Edinburgh, 2008.

[3] C. Dietz, "PriorityQueue", GitHub, 2019. [Online]. Available: <https://github.com/CollinDietz/PriorityQueue> . [Accessed: 07- Sep- 2019].

# 7 - Appendix

### 7.1 - Appendix A - Motor Controller

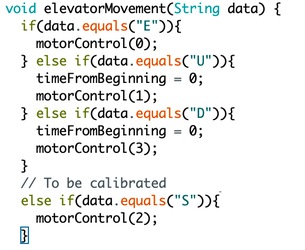


Figure 2 - Reading the data from control box and call the appropriate motorControl function



Figure 3 - motorControl function which takes the state from elevatorMovement and calls either moving up, down, stop or emergency stop.

### 7.2 - Appendix B - Priority Queue

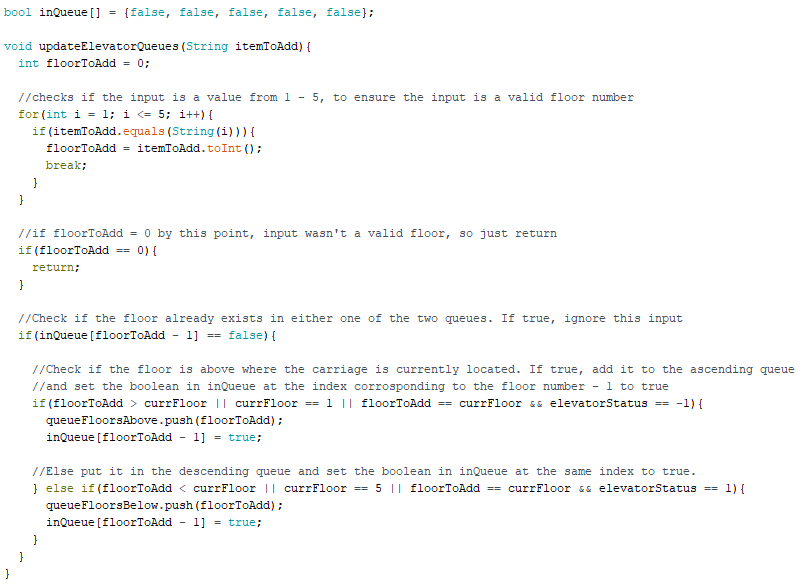


Figure 4 - updateElevatorQueues(), which takes a string input and updates the priority queues to include the string input as a new floor.

### 7.3 - Appendix C - Communications

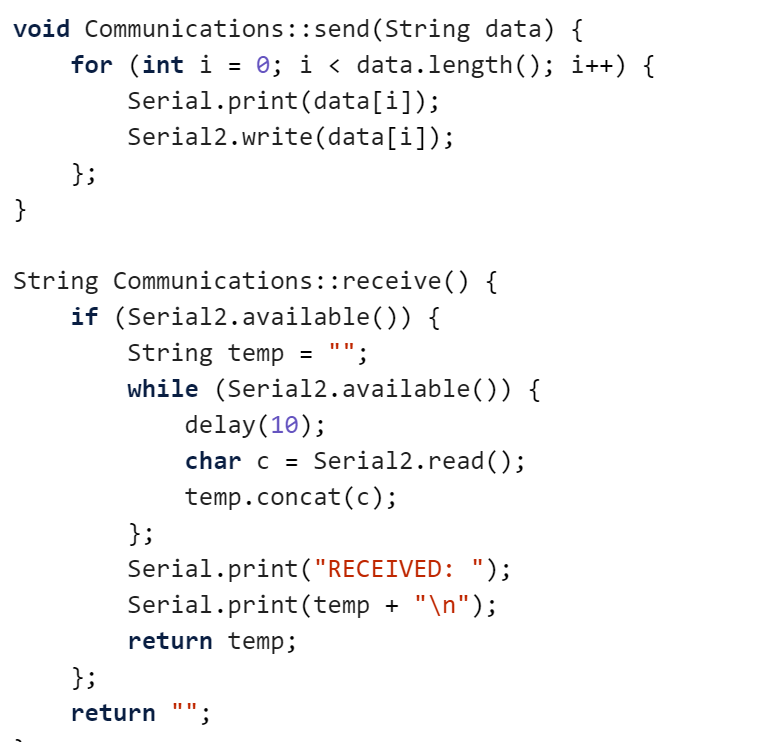


Figure 5 - Communications.receive() and Communications.send()

### 7.4 - Appendix D - Sensors

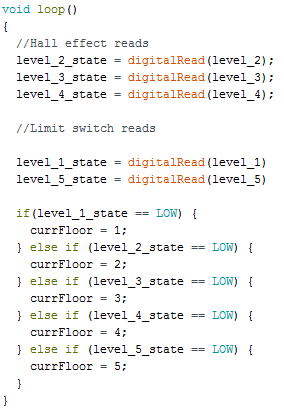


Figure 6 - Detects which floor the elevator is currently at within the loop.

### 7.5 - Appendix E - Path Finding

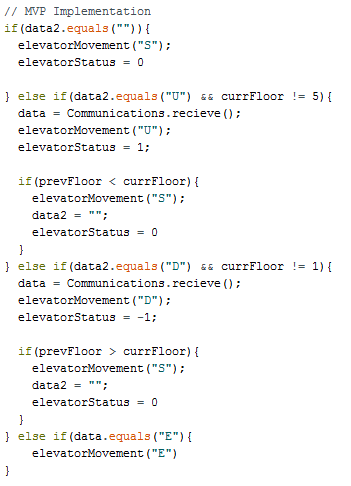


Figure 7 - MVP Path Finding implementation which calls for the lift to go up or down until the current floor is different to the floor that the elevator was at when it started moving.

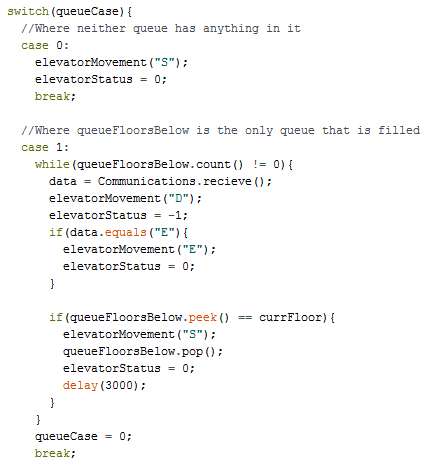


Figure 8 - Cases 0 and 1 of the Version 1 implementation, which checks for if the queues are filled or not and in what order they were filled in.

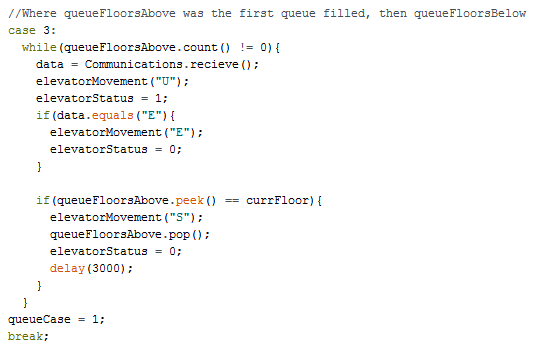


Figure 9 - Case 3 of the Version 1 implementation, where queueFloorsAbove is filled before queueFloorsBelow, so it’s required to exhaust that queue before moving to queueFloorsBelow.

### 7.6 - Appendix F - Requirements Document Feedback (See next page for PDF)