Project report, The Elevator Project TTK 4235 Embedded Systems

Jens Oskar Ramm-Pettersen
Thomas Kragerud
Gruppe 58

March 2022

Contents

1 Introduction 2

2 General architecture 3
2.1 Class diagram 3
2.2 State diagram 4
2.3 Sequence diagram 5

3 Module design 7
3.1 Elevio 7
3.2 Order 7
3.3 Timer 7
3.4 FSM 8

4 Testing 9
4.1 Unit testing 9
4.2 Integration testing 9
4.3 Requirements specification tests 9

5 Discussion 11
5.1 Modules and dependability 11
5.2 Scalability 11
5.3 Code quality 12

1 Introduction

During the course TTK 4235 Embedded Systems we have been developing a control system for an elevator with the requirements specifications given in the project assignment[1]. We have used the V-model[2] for development and testing and UML for design and documentation of our system. Finally we have used the programming language C and the environment Virtual studio code for implementation. This report will take you through the thought-process during the project and discuss some of the choices that we made and why.

[1]Appendix C, page 12 - 14, Embedded Systems Lab 1 Elevator
[2]Appendix A, page 9 - 11, Embedded Systems Lab 1 Elevator

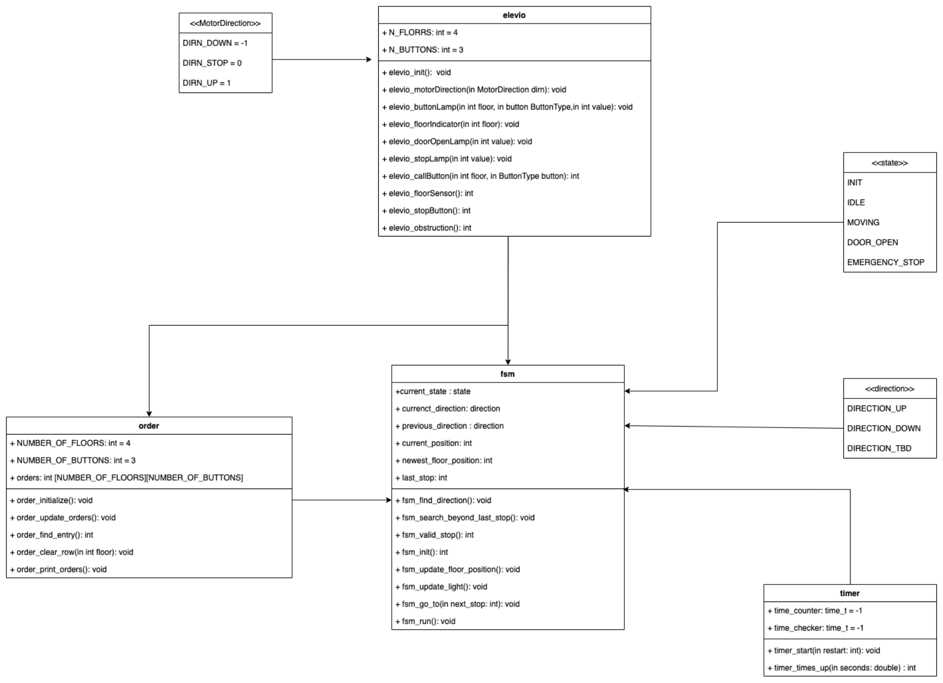
2 General architecture

Our programming philosophy for this project was from the start: make it work by making it simple. Therefore you will not see any C specific wizardry. Instead we chose to focus on making the code algorithmically clean, easy to read and navigate through and easily understandable for outside parties. The architecture consists of three different modules and they are used to communicate with the hardware through a driver.

2.1 Class diagram

The class diagram in Figure 1 below shows the different modules we have chosen for this project. We have a *order*-module that handles the queue of orders, a *timer*-module that is solely used for timing the elevator doors and how long they are open when at a floor. We also have a finite state machine with the name *fsm*. This module is the one that controls the elevators states and makes it act according to which state it is in.

The last module is the driver *elevio*, this is the module that communicates directly with the physical elevator, making it go up and down. It also controls all the lights on the hall buttons, cabin buttons and floor indicators.



<<MotorDirection>>
DIRN_DOWN = -1
DIRN_STOP = 0
DIRN_UP = 1
elevio
+ N_FLORRS: int = 4
+ N_BUTTONS: int = 3
+ elevio_init(): void
+ elevio_motorDirection(in MotorDirection dirn): void
+ elevio_buttonLamp(in int floor, in button ButtonType,in int value): void
+ elevio_floorIndicator(in int floor): void
+ elevio_doorOpenLamp(in int value): void
+ elevio_stopLamp(in int value): void
+ elevio_callButton(in int floor, in ButtonType button): int
+ elevio_floorSensor(): int
+ elevio_stopButton(): int
+ elevio_obstruction(): int
<<state>>
INIT
IDLE
MOVING
DOOR_OPEN
EMERGENCY_STOP
fsm
+current_state : state
+ currenct_direction: direction
+ previous_direction : direction
+ current_position: int
+ newest_floor_position: int
+ last_stop: int
+ fsm_find_direction(): void
+ fsm_search_beyond_last_stop(): void
+ fsm_valid_stop(): int
+ fsm_init(): int
+ fsm_update_floor_position(): void
+ fsm_update_light(): void
+ fsm_go_to(in next_stop: int): void
+ fsm_run(): void
<<direction>>
DIRECTION_UP
DIRECTION_DOWN
DIRECTION_TBD
order
+ NUMBER_OF_FLOORS: int = 4
+ NUMBER_OF_BUTTONS: int = 3
+ orders: int [NUMBER_OF_FLOORS][NUMBER_OF_BUTTONS]
+ order_initialize(): void
+ order_update_orders(): void
+ order_find_entry(): int
+ order_clear_row(in int floor): void
+ order_print_orders(): void
timer
+ time_counter: time_t = -1
+ time_checker: time_t = -1
+ timer_start(in restart: int): void
+ timer_times_up(in seconds: double) : int

Figure 1: Class diagram of the system and its four modules.

2.2 State diagram

Figure 2 shows a state diagram of the system with the five following states: INIT, IDLE, MOVING, DOOR_OPEN and EMERGENCY_STOP. The state diagram shows the behaviour of the system and how you can move from one state to another.

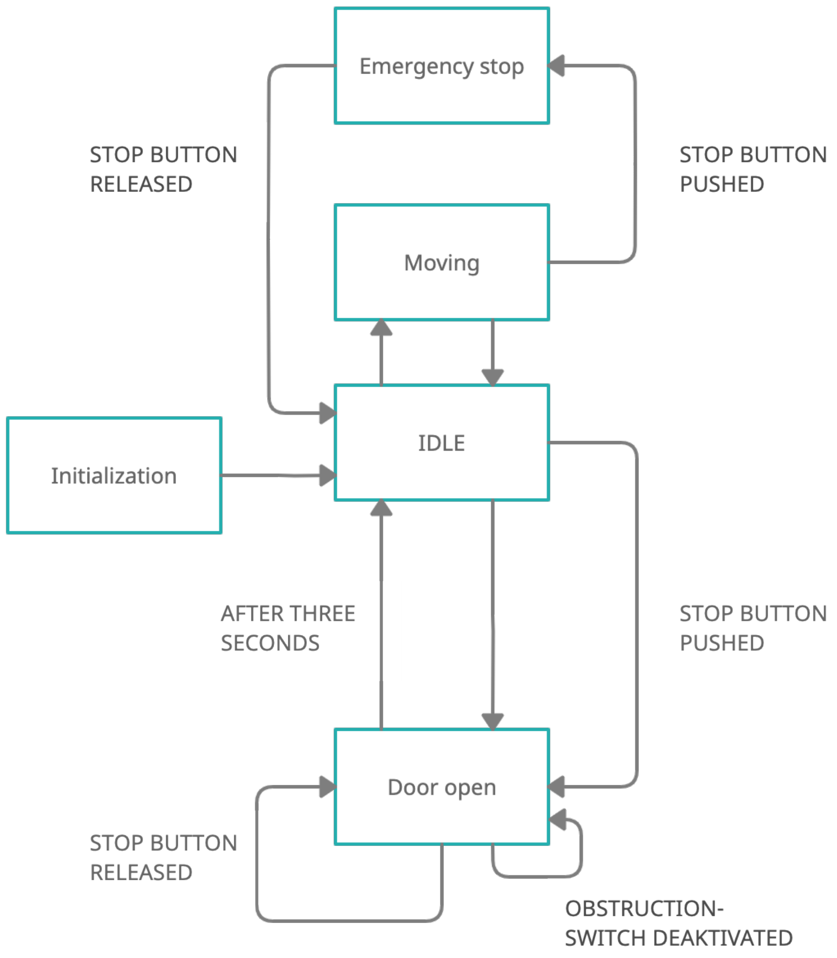


Figure 2: State diagram of the system

2.3 Sequence diagram

In Figure 3 we have made a specific sequence diagram showing the programs response to a certain sequence of events. The events are as follows:

1. The elevator is stationary at the 2nd floor with the doors closed.
2. A person orders the elevator from the 1st floor.
3. When the elevator arrives the person enters the elevator and orders the 4th floor.
4. The elevator arrives at the 4th floor and the person leaves.
5. After three seconds, the elevator door closes.

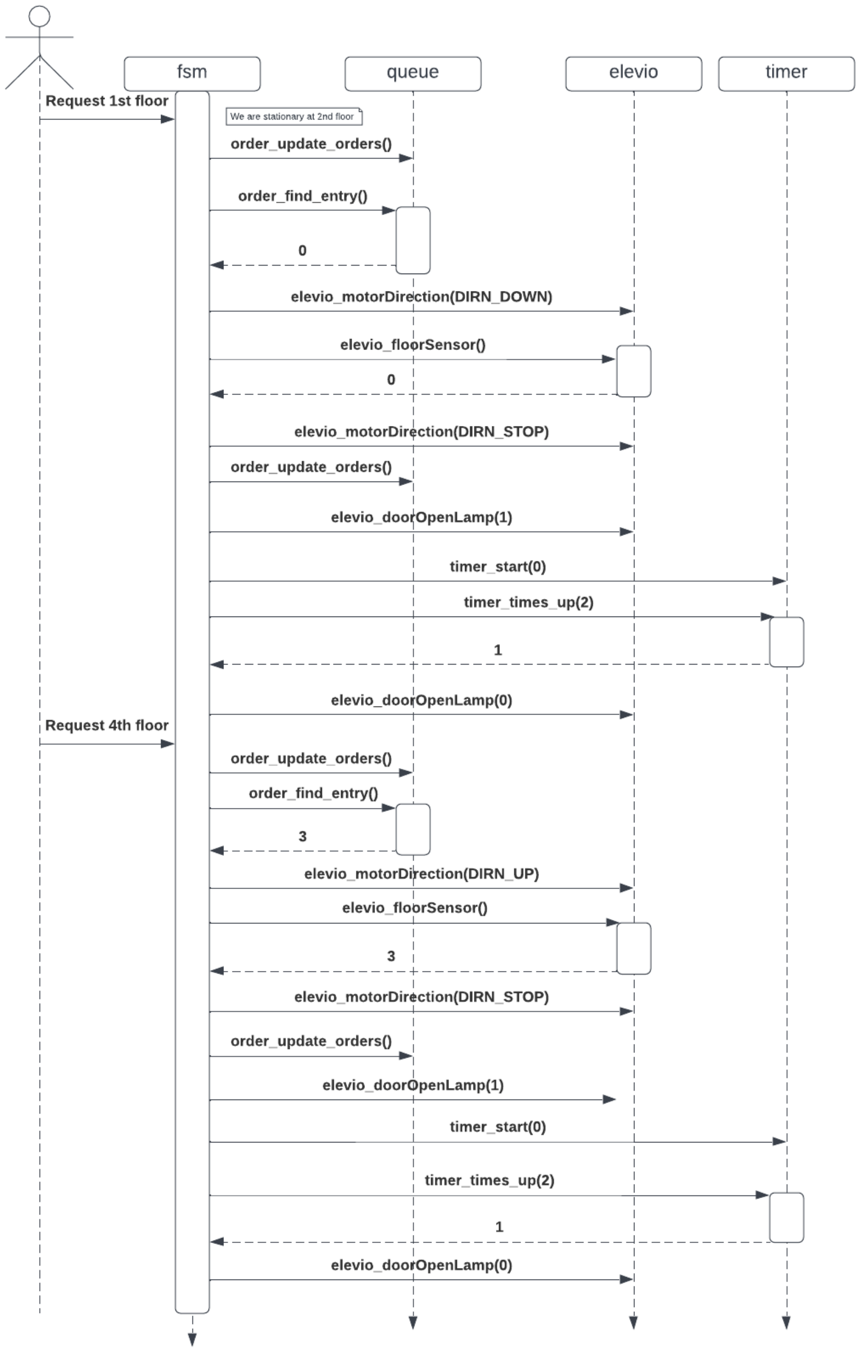


Figure 3: Sequence diagram following the given sequence

We feel that this architecture is a good choice because we use few modules with separate specific tasks. The code in itself is built up in a way that is easy to read and understand. There is nothing happening in the code that you cannot see. This makes troubleshooting a lot easier.

3 Module design

3.1 Elevio

This module was handed out together with the task. It consists of all the functions that we are using to affect the physical elevator. Among these are a function for setting the moving direction of the elevator(*elevio_motorDirection()*) and a function for using the elevator's hall-effect sensors to know which floor we are currently at(*elevio_floorSensor()*).

3.2 Order

We started by determining how to manage the "queue", which in our case isn't actually a queue, but a four by three matrix consisting only of zeroes and ones. The four rows in the matrix is representing the four floors we are operating on and the three columns represent the three types of buttons you can use to request the elevator to a floor. The three different buttons are upward hall button, downward hall button and cabin button. The matrix is represented in the table below with different orders in third and fourth. The up button in the fourth floor and the down button in the first floor is marked x because these buttons does not exist.

Floor	UP	DOWN	CABIN
1st floor	0	x	0
2snd floor	0	0	0
3rd floor	1	0	0
4th floor	x	1	0

We have named this module *order* bacuse it hold all the elevators orders. This module consist of simple functions with self-explanatory function names. We chose this approach because it is intuitive and easily scalable.

3.3 Timer

When we had the queue sorted out, we needed a timer to know how long the elevator-doors are supposed to be open and chose to make a separate module; *timer*, for dealing with this. The only two functions in this module is *timer_start(int restart)* and *timer_times_up(double seconds)*. The first one starts and restarts a timer, and the other one check if a certain amount of *seconds* has passed. The module is made this way because it makes it possible for the system to check for new entries while the doors are open, something that would not work if we used for -or while-loops that stalls the program.

Because we are constantly looping through the *fsm_run()*-function we ran into some problems with the timer. This is because we started a timer for every loop and therefore did not get the result we wanted. We solved this by taking in a parameter *restart* in the *timer_start()*-function to be able to restart it whenever we wanted. We also used an if-statement to tell the timer to start only when the time counter was -1 and when the time was up and we change states the time counter was given the value -1 once again. This makes the timer only run when we need it to and and stop when it is not used.

3.4 FSM

Lastly we needed a finite state machine to control the state of the system and make the elevator act accordingly. Therefore the last module in our design is named *fsm*. The *fsm*-module is the one that controls the whole elevator and its behaviour. Here we use a switch with different cases for every state. This switch is constantly looped through as long as the program is running. We chose this setup because it's structure offers good readability and it is easy to understand. It also makes it very easy to change the state back and fourth. The switch is located inside the function *fsm_run()*. Inside the module *fsm* we also have a lot of helper functions that are called on by the *fsm_run()*-function in different states. These function help the elevator know in which direction to move or which floors to stop at.

During the implementation of the *fsm* we used a lot of time to sketch different scenarios to know how to best serve a sequence of orders. In the process of finding the best way we had to scale the elevator to more floors and see what would have been the best solution with n floors instead of four. We boiled it down to three important functions that decide every stop the elevator makes. First we have the function *fsm_find_direction()*. This function does, as the name states, find the direction in which the elevator has to move to get to the next stop. When the direction is sat we use the function *fsm_valid_stop()* to see if there is any other valid stops on the way there. This function returns a one when there is either a person in the elevator that is going off at the floor that is being passed or there is someone at the floor that is going in the same direction as the elevator are traveling. The last function is called *fsm_search_beyond_next_stop()*. As the name suggests we want to search beyond the next stop to see if there is any other valid stops further away. This function finds the closest stop in the same direction and the stop that is furthest away in the opposite direction. What this means is that if the elevator is in the first floor and it is ordered downward in all the other floors, the elevator goes all the way up and gets the person in the fourth floor before turning and heading down.

4 Testing

After we were done implementing the code we used testing as the V-model suggests. We started by testing every separate module in a Unit Test. We then moved on to an Integration test and lastly tested our system against the requirements specifications.

4.1 Unit testing

To begin this part of the testing we started in the small by checking that our timer was working as expected. We created a loop that only broke when the function *timer_times_up(double seconds)* returned 1 and started a timer outside this loop. After trying with different values for *seconds* and verifying with a stopwatch we could be certain that the timer was working.
Moving on to the queue, we started by printing the whole queue-matrix while running the program and observed what happened to it when we made interactions with the elevator buttons. The values in the matrix changed from zero to ones as the prospective buttons were pressed. We then knew that the orders made by button presses were stores in our queue-matrix and this is all we wanted.

4.2 Integration testing

Integration testing is where we test the different modules together. We started by testing the lights on the elevator panels to see if they worked properly. By running the program and clicking the buttons we could see that the buttons lit up and stayed lighted as long as the order in that floor was not completed. The floor indicator lights was tested by driving the elevator up and down between first and fourth floor and we observed that the indicator changed as the elevator passed new floors.
The last part of the integration testing was the finite state machine. Making sure that the elevator behaved as we wanted and changed states correctly. To test all this we simply operated the elevator as we were customers and gave it numerous cases to see how it responded. The elevator handled almost all of the cases correctly, but we had to make a few changes in the code along the way.

4.3 Requirements specification tests

Before the FAT-test we had to be sure that the system worked according the the requirements specifications. We then went though all the specifications and tested them separately. All tests are performed with the whole program running through an infinite while loop calling the function *fsm_run()*.

O1 - We ran the program and saw the elevator travel down to the first floor and stop.
O2 - We ran the program while printing the queue and tried to order different

floors during initialization. The queue did not react to these attempts.
O3 - When starting the elevator above fourth floor the elevator worked as it should because we tell it to move down until it hits the first floor during initialization. When starting it below the first floor it went downward into the stop button at the bottom of the elevator track. The elevator does not take into account such unrealistic start locations.
H1 - For this specification we ran the program multiple times and gave the elevator new orders at different times and at an unnatural pace. The elevator still completed all orders, no matter when and how many we gave it.
H2 - While the elevator was on its way upward from the first floor to the fourth floor we ordered downward in the third and found that the elevator did not respond to this order on the way up, but rather when it was heading down again.
H3 - We tested this by printing the queue and pressing all the buttons in one specific floor. When the elevator reached this floor all the orders in the floor disappeared from the queue.
H4 - This point was tested by giving the elevator an order and observing what happened after the order was completed. The elevator did not move after completion of the order.
L1 - We made a bunch of orders from hall buttons and cabin buttons and saw the light stay lit until the order was completed.
L2 - This test was done together with test **L1** because when the order is completed there is no order at the floor and therefore it should not be lit. This was correct for our system.
L3, L4 and L5 - By making the elevator travel up and down between the floors we could easily see that all three of these specifications are working as specified.
L6 - For this specification we ran the program and gave it some random cases while pressing the stop button at different times. The stop button lights whenever it is pressed and never when it is not.
D1 - We sent the elevator to a random floor and timed the elevator doors manually with a stopwatch. The tests showed that the doors where open for approximately three seconds before they closed when stopping at a floor.
D2 - By sending the elevator to a random floor and observe its behaviour we could determine that the elevator doors were closed when it didn't have any uncompleted orders.
D3 - With the elevator stationary in a floor we pressed and hold the stop button. We then timed the elevator doors from release of the stop button and found that the doors stayed open for approximately three second after release.
D4 - With the elevator doors open we activated the obstruction button and waited a while. We then timed the elevator doors from when we deactivated the obstruction button and found that the doors stayed open for approximately three seconds after deactivation.
S1, S2 and S3 - During several rounds of testing we observed that the "door open"-light never went on between floors and the elevator was always standing still when the light was lit. The elevator was also never able to travel outside the defined area between floors one and four.
S4, S5, S6 and S7 - We printed the queue and tried pressing the stop button

at different times and saw that the elevator stopped immediately and the queue cleared all orders. We also found that the elevator ignored all attempts to make orders while the stop button was pressed. When the stop button was released the elevator stayed stationary, as expected.
R1 - By trying to activate the obstruction button when the elevator was moving we can with certainty say that this does not affect the elevator when the doors are closed.
R2 and R3 - During the whole time testing all the above specifications we never encountered a situation where the system had to be shut down, restarted or initialize one more time for it to work.
Y1 - For this specification we just used our imagination to come up with the most unusual behaviour and tested the elevator with very special order calls and emergency stops. The system handled this just fine.

5 Discussion

5.1 Modules and dependability

It is without a doubt possible to dissect the problem into even more modules, we did contemplate on making designated modules for determining the next stops and setting the lights. Yet, only moving the functions in a different file would not have changed much as the dependencies would have worked both ways, essentially counteracting the point of modularizing in the first place. Thus, one would need a top-level logic redesign. Nevertheless, given the projects limited size we did not deem this necessary. We did however experience how difficult it is to work simultaneously on the same module. And on the contrary, we observed a productivity increase when we first established what different modules and specific functions should take in and return.

5.2 Scalability

Regarding the sequence in which different order are fulfilled, we are confident that our algorithm preforms optimally. However, when scaling to a taller building with additional floors, one would need to weigh the cost of always searching beyond the tentative last_stop looking for up and down buttons. One solution could be to record the floor which first ordered an elevator and then only search for hall orders in say 5 floors up and down from the initial order until that floor is expedited. Looking ahead to the next laboratory exercise in *TTK4145 Real-Time Programming* were we are making the same system for an unlimited amount of elevators, we believe much of our code can be reused by threading our system and including a cost function that communicates with the different elevator threads.

5.3 Code quality

To make life easy both for us and external readers, we embraced an explicit (pythonic - simple is better than complicated) coding style, with as little as possible hidden from the reader. We did for instance spend a lot of time ensuring that we only used a single top level while loop. Drawing inspiration from game design, we viewed the loop frequency as the refresh rate of a game frame, updating everything from lights to orders multiple times a second. This was done to make it possible for external readers to easily follow the thread of sequential logic. Following this philosophy, we explicitly change the states within the switch case statement of fsm_run() and chose to use additional if statements instead of contracting it in to one complicated line.
On the contrary we did commit one of the deadly sins of programming, namely Blackbox programming. In retrospect we could definitely have created a struct with direction, end stop, current position etc. But with limited experience in c programming going into this project we chose to stick with our motto make it work by making it easy. To circumvent the classic pitfalls of Blackbox programming we have rigorously documented the functions, list all variables that are subject to change and used names reflecting functional cohesion, where one function only has one specific task. In the case of the latter, we were initially reluctant to create the function fsm_update_lights() since we could save multiple lines by updating the lights in order_update_order(). However, we realised that this would be very confusing for external users if something was to change in elevio.