Norwegian University of Science and Technology

TTK4235 - REPORT

Embedded Systems

Elevator Project

Group members
Elias Fuglestrand
Audun Svinø Seeberg

Group number: 15

February 27, 2020 (Spring 2020)

https://www.ntnu.edu/studies/courses/TTK4235

NTNU
Norwegian University of
Science and Technology

Contents

1 Introduction

This project is based around an elevator model, consisting of a motor, four floor sensors, and a control unit made to simulate the inside and outside panels of an elevator. The goal is to develop a control system for the elevator, using development tools like git for source control, GDB and Valgrind for debugging, and Doxygen for documentation. The project is to be developed according to the "V-model", and various UML-diagrams are used in early development to lay out the architecture and module designs before implementation.

2 Architecture design

2.1 Class diagram

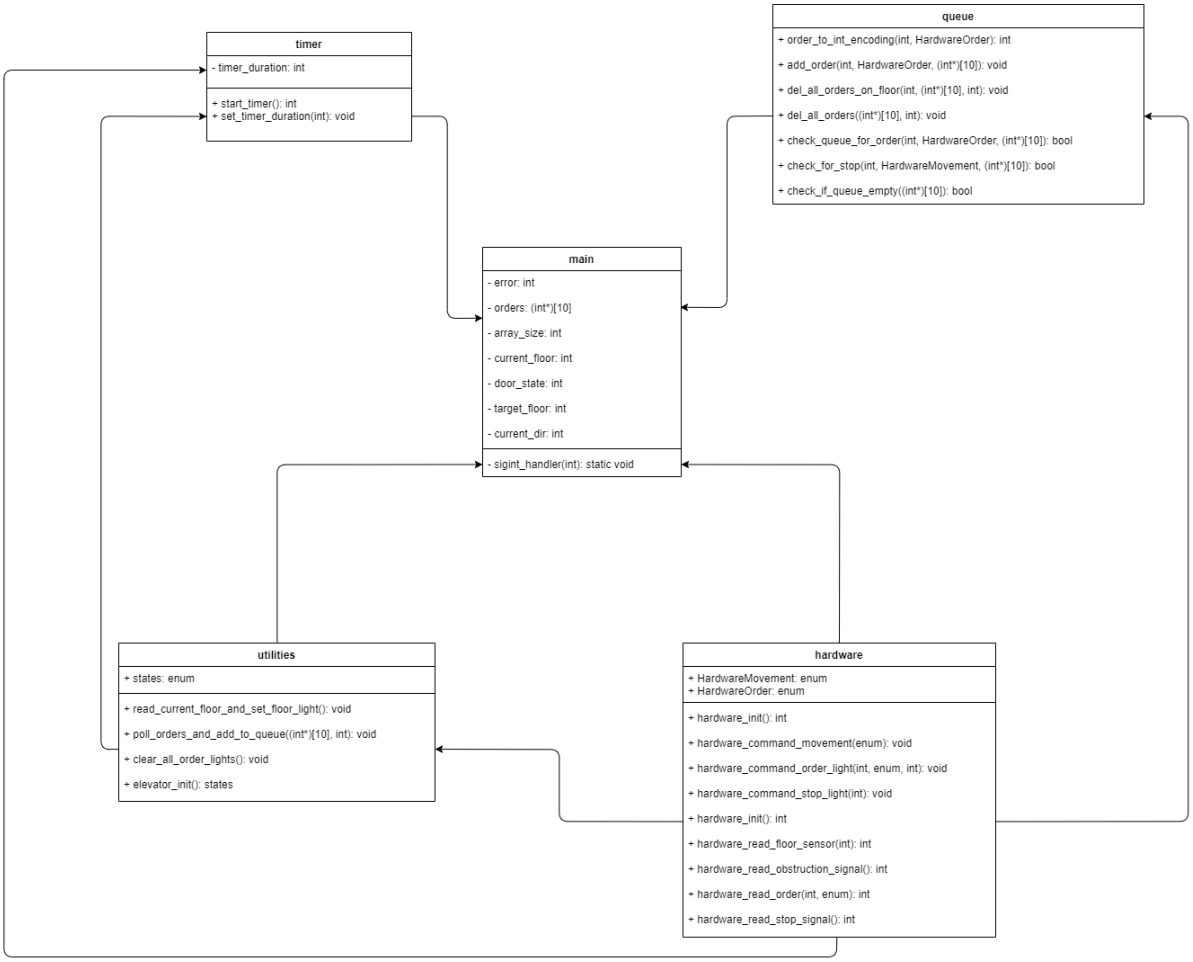

Figure 1: Class Diagram

The class diagram shows the system of modules. We aim at keeping the number of modules as low as possible as well as keeping the dependencies between modules at a minimum.

We want to have a separate module for utilities in order to simplify interaction with hardware, as the hardware module is a part of the driver, and should thus be regarded as a "black

box". The idea is to group hardware functions that are used exclusively in combination, into a single utilities-function, and as a result making hardware interaction easier.

Some of the specifications of the elevator requires a timed delay for the door, which explains the choice of a separate timer-module.

Furthermore, we want to implement a queue-system for the orders, in order to decide which floor the elevator should go to. This is implemented in the queue-module.

We choose to place the finite state machine that is to control the elevator inside `main` instead of in a separate module. This is done to make it easier to read the full procedure of initialization and error-checks before the elevator enters the FSM.

2.2 Finite state diagram

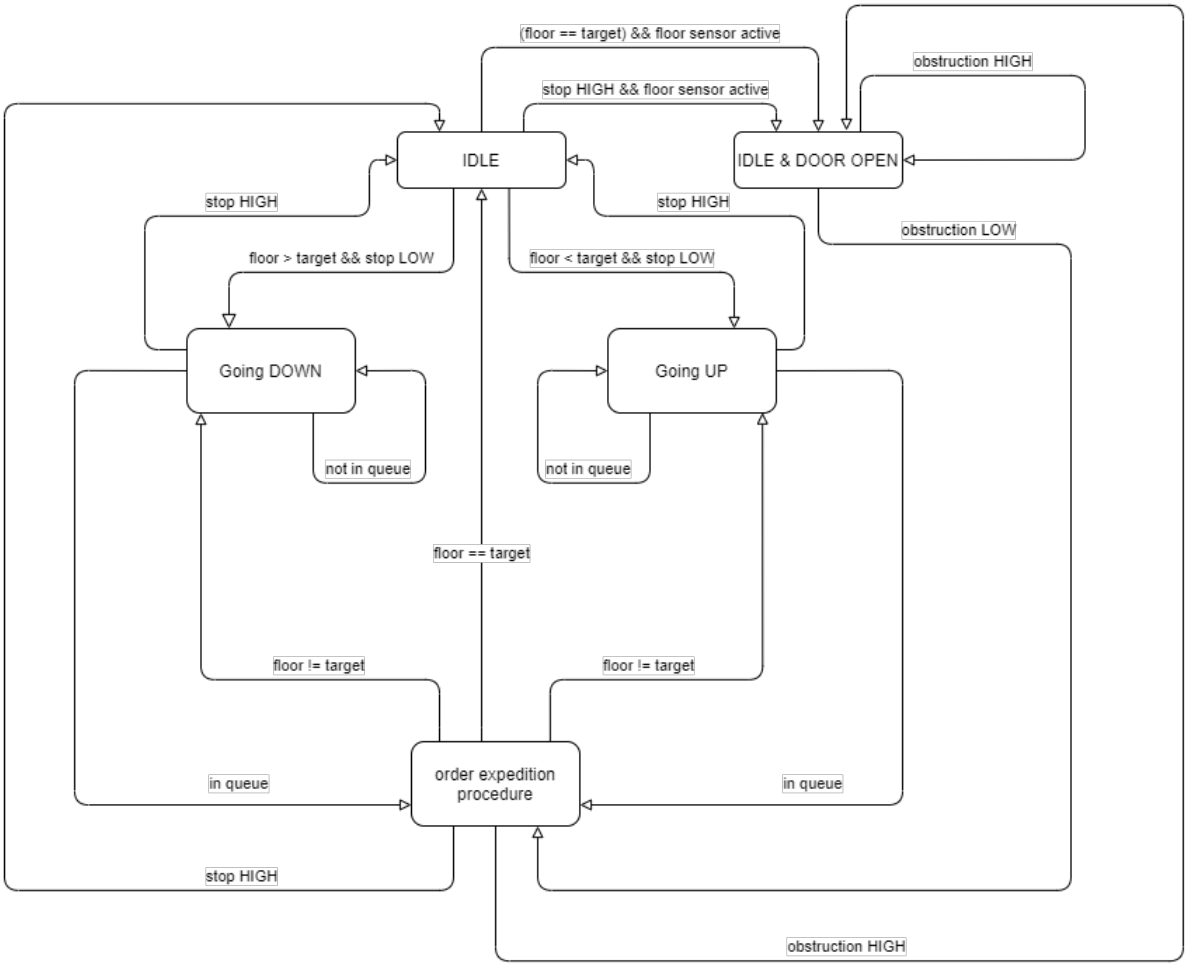

Figure 2: Finite State Diagram

The state "IDLE DOOR OPEN" in the diagram is not an actual state in our implementation of the finite state machine, but rather an abstraction of the logic implemented in "order expedition procedure" and "IDLE". This could, of course, be implemented as it's own state, and you could argue that a separate state makes the code more readable. However, the implementation of this state into "IDLE" and "order expedition procedure" is fairly trivial, so we choose to leave it there.

2.3 Sequence diagram

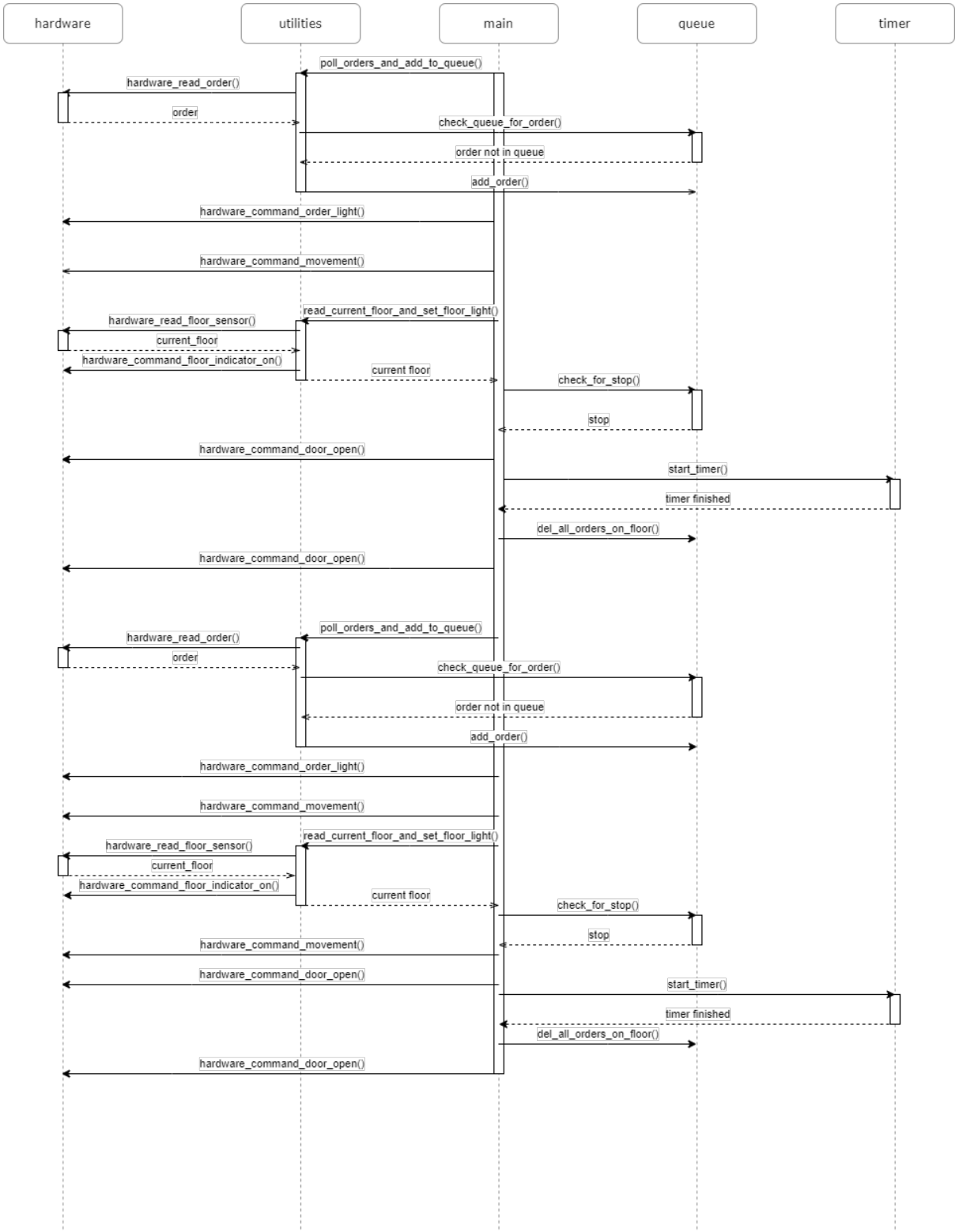

Figure 3: Sequence Diagram

The diagram in Figure 3 shows a sequence where the elevator is idle at the second floor with the doors closed when a person makes an order from the first floor. When the elevator arrives the person gets in and orders the fourth floor, rides the elevator there and gets out. At the end the door closes after three seconds.

3 Module design

3.1 `hardware`

The hardware-module is a part of the project files that are handed out, and as a result, no further implementations are added. The functions and enums are used in several other modules, however only the doxygen-documentation from the `.h`-file is used to understand how to make use them, so the hardware-module can be regarded as a sort of "black box".

3.2 `utilities`

The utilities-module contains functions for polling orders and reading the floor sensors. As previously explained these functions are just groupings of other hardware and queue functions. The module also contains the functions for the initialisation of the elevator along with the enum containing different states. In our design all floor sensors are read at the same time sequentially, therefore it made sense bundling these calls to hardware in one functions. The same is the case for reading order buttons. The higher level of abstraction also increases the readability of the code.

3.3 `queue`

The queue-module is implemented using an array of constant size to store orders. We encode every order to an int ranging from 0-11 and when an order is made, add the corresponding int to the first available slot in the array. To make sure that all orders are expedited in the correct order we rearrange the array, pushing all empty slots to the back each time orders are deleted. The module also contains functions to check if there are any orders in the queue, and to check if the elevator is to stop given a floor and direction.

This implementation of the queue ensures that the orders are expedited in the correct order without use of a timer. The encoding of orders as ints eliminate the need for structs and this together with the constant size array makes debugging and handling memory easier.

3.4 `timer`

The timer module is implemented with two functions; `set_timer_duration()` and `start_timer()`. As all our timers are 3 seconds long, we could easily manage without the `set_timer_duration()` function, however we find it good practice to be able to change the timer-duration if we are to use the code in future projects.

In the `start_timer()` function, we have also implemented some functionality to be able to take orders and read stop- and obstruction-signals while the timer is running. The function returns `0` when the timer is finished, and is in some cases used directly in `hardware_command_door_open()`, to close the elevator door when the timer finished. This explains the choice to add hardware functions inside the timer, as waiting for it to give a return value would halt the program entirely for 3 seconds without them.

3.5 `main`

The main module is where we implement our finite state machine, shown in Figure 2. This implementation makes use of an enum from `utilities` called `states`. It's a fairly normal FSM implementation, where we use a switch, and have our states as cases in the switch. For each state we implement the responses to different events, for example setting the FSM in "GOING UP" in response to an order to the 4th floor when in the 1st floor. The switch is run inside a `while(1)`-loop, and will not terminate unless we program is terminated.

4 Testing

4.1 Unit testing

For the simplest functions, i.e. the timer, testing is fairly problem-free. A simple `printf()` and observation of output upon execution will assure us that our timer finishes after the requested time. This practice generally holds for functions not directly related to hardware, but falls short when functions and modules become more dependent. This way of testing, as well as some use of GDB on separate functions is used mainly in `timer` and in `queue`. We test our `queue`-functions by disabling the `hardware_command_movement()`-functions and driving the elevator manually by pushing the car. This makes it easier to read variables in GDB while running. Reading the queue in GDB, adding an order, and reading the queue again is a simple way to debug, but proves very effective.

4.2 Integration- and system testing

Integration testing is done by systematically going through the system specifications and making sure the elevator fulfills the requirements.

We start the elevator in an undefined state, i.e. between two floors. When the program then executes, the car starts moving downwards, and while it does we mash the order buttons. The elevator stops when it hits a floor sensor, and no orders are taken. This assures that specification O1, O2 and O3 are met.

Setting the elevator in the 1st floor, and making an inside order to 2nd floor, a downward order on the 4th floor, a downward order on the 3rd floor, and an inside order to the 4th floor, in that order(as in sequence), tests specifications H2, H3, and H4, L1, L2, L3, L4, L5, D1, and D2. Observing the elevator and control panels, we see that the correct order lights are set when we make the orders. The elevator starts moving upwards until it stops on the 2nd floor, where the floor light is set. The door is opened for 3 seconds, then closes, and the order light is turned off. When the door is closed the elevator moves upwards, but does not stop at the 3rd floor. The 3rd floor light is lit, and continues to be lit until the elevator proceeds to hit the 4th floor, where it stops, opens the door for 3 seconds, then clears both the inside order and outside order. It proceeds to the 3rd floor where it opens the door for 3 seconds, clears the order, and closes the door. It then stands idle indefinitely. This assures the aforementioned requirements are met.

Activating obstruction with door open and with door closed we observe that the door remains open when obstruction is active if the door was already open, and does nothing when the door is closed. When making orders with obstruction active the orders are taken but the door remains open. This assures that specification D4, S1 and R1 are met.

Adding orders and activating the stop button while at a floor and while in-between floors we observe that the elevator never moves while stop is active, and that all orders previously taken are deleted. Additionally, no orders can be made while holding the stop-button. The door never opens while not at a floor. We also ran the program through Valgrind to check for memory leaks, and found none. This together with the previous tests assures all requirements are met.

5 Discussion

As a whole, we view our solution as a an easy to read implementation of a fully working elevator. Of course, we are not totally objective in this matter, and we know that other popular implementations, especially in the queue, also solve the problem well. To criticize our own solution, the main-module may seem messy to some. To improve readability, we could have moved our implementation of the FSM to a separate module, and called the different "state-functions" from main. This provides better readability in main, however the functions from the FSM would say little about what is actually going on, as a single state has a lot of functionality built in. We have aimed to make our function names as self explanatory as possible, and explaining all the things a state does in a single function-name would make for far worse readability.

As for the queue, one criticism could be that the encoding of orders makes it less obvious which exact order is made, as they are encoded to `iint`s that require you to know the encoding algorithm we chose, as well as the `enum` which specifies the order type. In this case we had to make a compromise between readability and implementation-simplicity. We concluded that the improvement in the simplicity of the queue outweighed the lack of readability, and instead tried to compensate by adding extra thorough documentation in this part of our implementation.

Commenting on the timer, as of now it's fairly customized for this exact project.
Ideally we would have a timer that simply timed, and place the other functionality elsewhere, but we have not yet found an implementation that would solve this.