Firmware Design
Project Report

Elevator Controller

Université Saint-Joseph de Beyrouth
Faculté d'ingénierie et d'architecture
Institut national des télécommunications et de l'informatique

January 9, 2025
Submitted by

Ghady Youssef
ghady.youssef@net.usj.edu.lb

Antoine Karam
antoine.karam3@net.usj.edu.lb

Contents

Abstract

This report presents the design and development of an elevator control system utilizing microcontrollers to manage various hardware components, including temperature sensors, real-time clocks, and motor controllers. The system's primary objectives are to optimize elevator operation through efficient scheduling, ensure safety during motor operations, and manage floor and cabin displays. The project addresses challenges such as minimizing passenger waiting times, preventing unsafe motor behavior, and ensuring reliable system performance. The integration of hardware and software components creates a robust elevator control system with potential applications in other safety-critical embedded systems.

Chapter 1

Introduction

This report outlines the design and development of an elevator control system using microcontrollers to manage hardware components like temperature sensors, real-time clocks, and motor controllers. The system aims to optimize elevator operations by implementing an efficient scheduling algorithm, ensuring safety during motor operations, and managing floor and cabin displays.

The project addresses many challenges, including minimizing passenger waiting times and preventing unsafe motor behavior. It integrates hardware components with software interfaces and real-time scheduling algorithms, creating a reliable and efficient system for elevator operation.

Chapter 2

Firmware Architecture

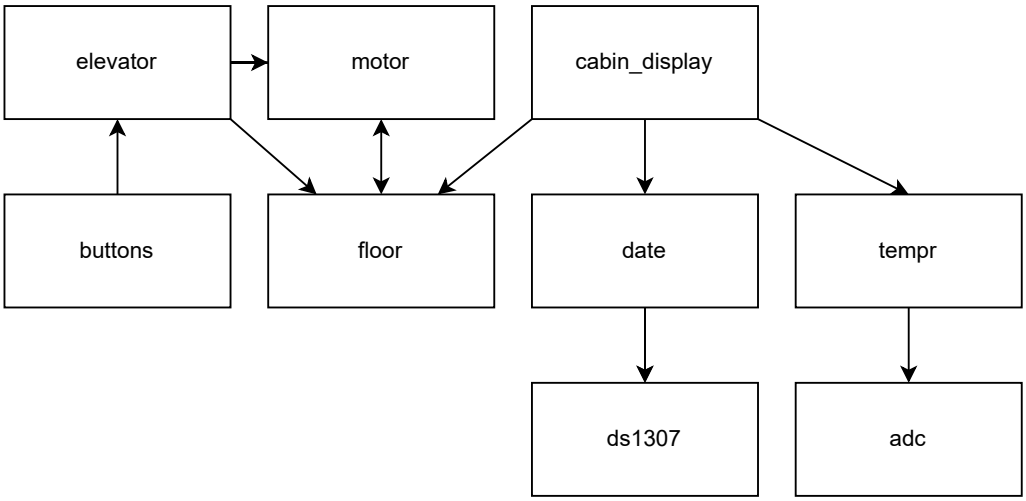

Figure 2.1: Block diagram representing the firmware architecture

Chapter 3

Drivers

3.1 Temperature: LM35

The **LM35** is a temperature sensor used to display the temperature inside the cabin. Its implementation is straightforward, as it has already been implemented in another project. The LM35 is connected to the system's analog-to-digital converter (ADC), utilizing the ADC driver.

The temperature API provides two main functions: one to directly get the temperature and another to get it in BCD format, allowing easy display on the cabin's 7-segment displays, limited to 2 digits.

3.2 Real-Time Clock: DS1307

In order to display the date and time inside the cabin, we used the **DS1307** Real-Time Clock. We referred to the datasheet to implement the required driver code, exposing a `read_reg` function to read from the internal chip registers. The `date` API contains utility functions which allows us to retrieve the date, month, year and time data. The driver code was implemented using I^2C, in order to transfer the regitser data from the chip to the microcontroller.

Chapter 4

Display

4.1 Floor Display

The floor displays on each floor and inside the cabin show the current status of the cabin's floor. The `floor` API monitors through the `floor_state_monitor` task the swtiches present on each floor and updates the `current_floor` global state. The current state can be accessed through the `get_current_floor` function.

4.2 Cabin Display

The cabin display shows the time (HH:MM), date (DD MM), and current temperature (TT °C) in sequence, cycling every 10 seconds. It utilizes four 7-segment displays, paired with a separator display in the middle.

The temperature is retrieved in BCD format using the temperature API, while the time and date are fetched from the data API, also in BCD format, facilitating the display on the 7 segments.

The cabin display operates as a low-priority task running every 1 second. This frequency ensures the display separator blinks in a heartbeat-like manner when showing the time and allows timely updates from the APIs in case of any changes.

Chapter 5

Motor

The motor is a crucial component of the elevator system, responsible for moving the cabin between floors.

5.1 Motor Operations

The motor API provides two main functions. One function is used by the elevator scheduler, which is discussed in the next section, and another to determine the motor's current direction. The motor direction is important for both the scheduler and the floor display, as it helps the scheduler choose the next floor and it ensures the correct arrow is shown for the cabin's movement.

The motor determines its movement based on the current floor provided by the floor api and the target floor provided by the scheduler. It listens to the switches to manage speed: it accelerates when leaving the start floor, decelerates as it approaches the destination, and stops once it reaches the target.

5.2 Safety Considerations

The motor is designed with passenger safety in mind, ensuring that it does not blindly follow scheduler commands that could lead to unsafe behavior.

A key safety feature prevents the motor from suddenly changing direction. If the motor is already moving in one direction and the scheduler requests a change to the opposite direction, the motor will ignore the new request until it reaches its current destination and becomes idle. This ensures that the motor does not perform abrupt or unsafe directional changes, maintaining a safe ride.

In cases where the scheduler reschedules the motor to a new floor in the same direction, the motor will accept new the scheduler request. This design ensures a balance between responsiveness and safety.

Chapter 6

Scheduler

6.1 Responsibility

The scheduler's primary responsibility is to determine the next floor the elevator should move to based on the current position, the direction of the motor, and the floor and cabin requests. It ensures that the elevator operates efficiently by selecting the most optimal floor to serve, maximizing resource utilization given that there is only one elevator in the system.

Additionally, the scheduler is responsible for ensuring safe operation by avoiding abrupt direction changes. It prevents selecting a new floor that requires a sudden shift in direction, ensuring predictable elevator movement for the safety of the passengers.

The elevator scheduler processes incoming requests from the cabin and from each floor. It selects the next target floor for the elevator, taking into account priorites, and staying in the same direction, while minimizing long trips if possible.

To manage these requests, the scheduler stores two queues: one for requests to go up and another for requests to go down.

6.2 Elementary algorithm

1. When the elevator is **completely** idle, it is free to go in any direction.
2. When the elevator is going in a direction, it must preserve this direction until the respective queues becomes empty.

6.3 Reordering the queues

When we reach the destination (the top of the queue) and dequeue it, we have to reorder both queues to ensure the invariant of the `queue_t`'s first and second are preserved. That is, the first part contains the floors which are *(resp. below, above)* the current floor for the *(resp. down, up)* queues and the second part contains the floors which are *(resp. above, below)* the cabin floor for the *(resp. down, up)* queue.

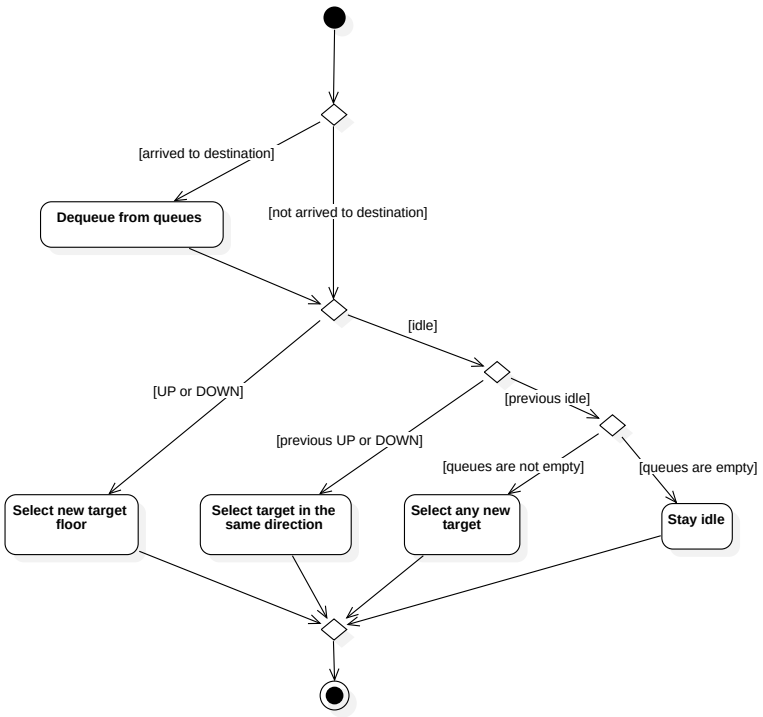

Figure 6.1: Activity diagram showcasing the elevator scheduling algorithm

Chapter 7

Development Workflow

7.1 Version Control

To complete this project, we made great use of developer tools like Git. It allowed us to easily collaborate and distribute the tasks between the team members. We also heavily relied on GitHub Issues to assign tasks to each of the team members. When each issue is completed a pull request is created by the assignee and reviewed by the other member to ensure that both members are aware of the changes made to the codebase. This was a great opportunity to practice modern software engineering methodologies.

We seperated our work into feature (or fix) branches which were constantly created from the `dev` branch. Once each feature is completed, it is then merged back into main. We primarily use `dev` for our workflow while `main` acts as a mirror of `dev` to push new releases.

7.2 Continuous Integration (CI)

To ensure the codebase remains clean and free of unnecessary style issues, we used `clang-format` with a specific configuration to keep the styling consistent. Additionally, we used `cppcheck` to check for any additional linting warnings or errors that we could have missed during development.

Furthermore, we developed CI pipelines using GitHub Actions in order to automate building releases, style checks, and generate documentation using `doxygen`. Since most off the codebase already contains comments in the `doxygen` format, it made complete sense to have a pipeline dedicated for generating documentation. The live documentation is a available at https://ghaadyy.github.io/elevator-controller/.

Chapter 8

Challenges

One of the core challenges faced during the development of this project was to effectively design the elevator's scheduling algorithm. We wanted to create an efficient algorithm that maximizes the number of trips in the same direction.

We had to thouroughly test and debug in order to make sure the algorithm works *correctly*. When we were in doubt during the development, we made great use of the debugger to step through the code. Most of the time, the errors were logical and not related to the compilation of the code itself. Moreover, we decided to work on the elevator scheduler in a pair-programming fashion. Since this was a difficult task, reasoning throughout this problem together allowed us to progress much faster.

Every time we solve a bug, another one arises, and we have to carefully analyze and document every possible interaction the scheduler might encounter. This helps us process all edge cases and ensure a reliable scheduler. Throughout this process, we've had to revise the elevator design multiple times. Initially, we stored all requests in a single queue, but we then switched to using separate queues for external and internal requests. Later, we merged the cabin and floor requests into two queues, one for requests to go up and another for requests to go down. Eventually, we divided these queues into two parts to further optimize the process.

Chapter 9

Future Work

9.1 Error Detection

Currently, there is no mechanism in place to anticipate what might happen if the power suddenly fails or if the motor stops unexpectedly. We need to implement error detection and handling procedures to address these potential failures, ensuring a safe ride for passengers under all circumstances.