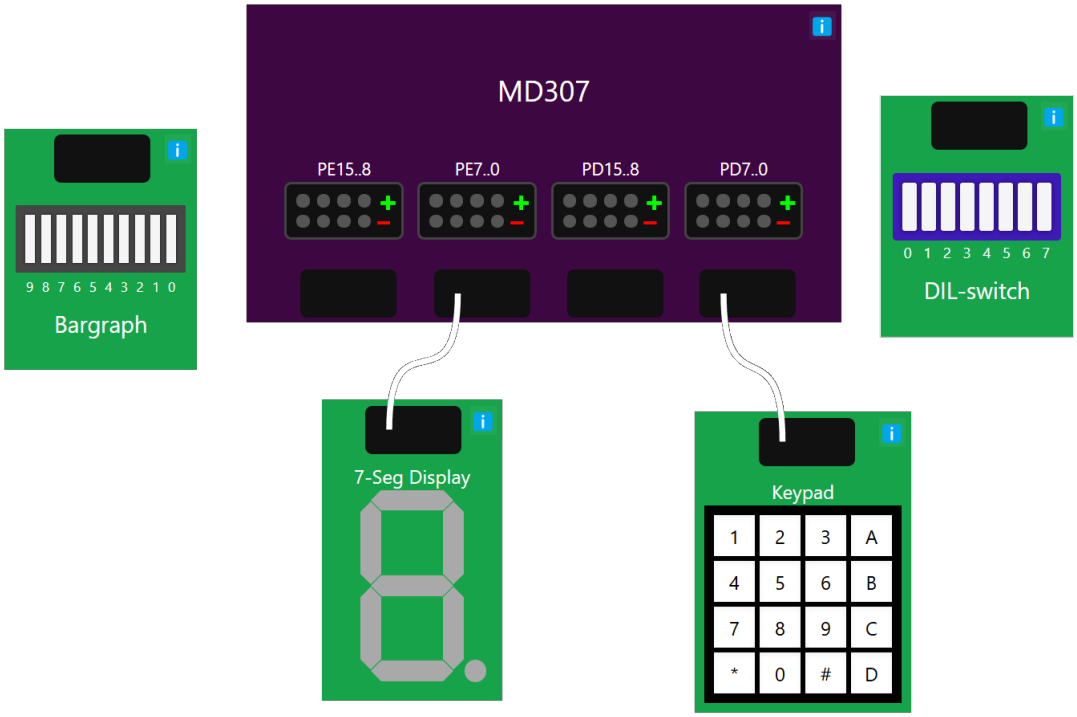
CHALMERS | UNIVERSITY OF GOTHENBURG



Web Based I/O Simulator for Education in Machine-Oriented Programming

Bachelor's thesis in Computer Science and Engineering

Omar Ahmed, Niklas Andersson, Alexander Kjellberg, Andreas Nilsson, Cecilia Nordén Elgh, Oskar Ranhage

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025
www.chalmers.se
www.gu.se

Bachelor's thesis 2025

Web Based I/O Simulator for Education in Machine-Oriented Programming

Omar Ahmed
Niklas Andersson
Alexander Kjellberg
Andreas Nilsson
Cecilia Nordén Elgh
Oskar Ranhage

UNIVERSITY OF
GOTHENBURG

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Web Based I/O Simulator for education in Machine-Oriented Programming

Supervisor: Erik Sintorn, Elias Hällqvist, Department of Computer Science and Engineering
Examiner: Arne Linde, Department of Computer Science and Engineering

Bachelor's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Omar Ahmed, Niklas Andersson, Alexander Kjellberg, Andreas Nilsson, Cecilia Nordén Elgh, Oskar Ranhage

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Machine-Oriented Programming course at Chalmers University of Technology and the University of Gothenburg introduces students to low-level programming through microcontroller-based laboratory exercises. During the course, physical access to hardware is limited to scheduled lab sessions, which restricts opportunities for practice. To address this issue, a simulator is used to simulate the microcontroller unit (MCU) and the connectable Input/Output (I/O) units that are used in the course.

This bachelor's thesis presents the development of a web-based simulator for the I/O units that interfaces with the existing MCU simulator via WebSockets. This I/O simulator is developed separate from the MCU logic, with the goal of making it platform-independent as well as more usable, maintainable and accessible. It supports various I/O units such as switches, bargraphs, 7-segment displays and keypads.

The application was evaluated through user testing with students who have previously taken the course. The overall reception was positive in terms of usability, visual design, and its potential as an educational tool for understanding and experimenting with machine-oriented programming.

Keywords: Simulator, Machine-Oriented Programming, Education, Web Development, User Interface Design, Web Application, Svelte, WebSocket

Sammandrag

I kursen Maskinorienterad Programmering på Chalmers tekniska högskola och Göteborgs universitet introduceras studenter till maskinnära programmering genom mikrokontrollerbaserade laborationstillfällen. Under kursen är tillgången till fysisk hårdvara begränsad till dessa tillfällen, vilket minskar möjligheterna att öva. För att lösa det här problemet används en simulator som simulerar mikrokontrollern (MCU) och dess anslutningsbara Input/Output (I/O) enheter som används i kursen.

Detta kandidatarbete presenterar utvecklingen av en webbaserad simulator för I/O enheter som interagerar med den existerande MCU simulatorn genom WebSockets. I/O simulatorn är utvecklad separat ifrån MCU logiken med målet att göra den plattform-oberoende och mer användarvänlig, mer underhållsvänlig och lättillgänglig. Den stödjer olika I/O enheter såsom switchar, lysdiodramper, 7-segmentsdisplayer och knappsatser.

Applikationen utvärderades genom användartester med studenter som tidigare gått kursen. Generellt sett var mottagandet positivt, både gällande användarvänlighet, visuell design och dess potential som ett utbildningsverktyg för att förstå och experimentera med maskinorienterad programmering.

Nyckelord: Simulator, Maskinorienterad Programmering, Utbildning, Webbutveckling, Webbapplikation, Svelte, WebSocket, Gränssnittsdesign

Acknowledgements

We would first and foremost like to thank our supervisors Erik Sintorn and Elias Hällqvist for their engagement and support throughout the project. We would also like to thank all the students who have participated in interviews and user tests, as well as FlashIT for letting us borrow their camera to take pictures of the laboration equipment. Finally, we would like to thank Chalmers Univeristy of Technology and University of Gothenburg for all the resources and advice that has been a great help during the project.

Omar Ahmed, Niklas Andersson, Alexander Kjellberg, Andreas Nilsson, Cecilia Nordén Elgh, Oskar Ranhage, Gothenburg, May 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
CSS	Cascading Style Sheets
DOM	Document Object Model
DIP	Dual In-Line Package
GDB	GNU Debugger
GND	Ground
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HTML	HyperText Markup Language
I/O	Input Output or Input/Output
IDE	Integrated Development Environment
ISA	Instruction Set Architecture
JSON	JavaScript Object Notation
LCD	Liquid Crystal Display
MCU	Microcontroller Unit
MD307	MikroDator 307
MD407	MikroDator 407
MOP	Machine-Oriented Programming
RISC	Reduced Instruction Set Computer
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
UI	User Interface
VDD	Power Supply Pin (Voltage Drain)
VSCode	Visual Studio Code

Contents

List of Figures

1
Introduction

In computer science and engineering, an understanding of the low-level mechanics of computer systems is important to bridge the gap between abstract theory and the behaviour of computing hardware. The Machine-Oriented Programming (MOP) course at Chalmers University of Technology and the University of Gothenburg introduces students to low-level programming and small embedded systems, using programmable microcontroller units (MCUs) during laboratory sessions [1], [2]. A new MCU is currently under development that will eventually replace the current model.

1.1 Background

Since hardware is only available to students during physical laboratory sessions, practice hours are limited. A simulator called SimServer is created to address this limitation, enabling students to learn and practice in a simulated environment on their own computers outside of laboratory sessions. SimServer supports various course-specific MCUs, including the currently used MikroDator 407 (MD407), and the new MikroDator 307 (MD307). SimServer is compiled into a single executable file, but consists of several different simulation modules: the MCU simulation, the connectable peripheral Input/Output (I/O) unit simulation, and a Graphical User Interface (GUI).

Compiling all these parts into one executable file introduces a limitation that makes the system more difficult to maintain, as updates and bug fixes must be distributed by manually downloading and recompiling new versions. Another limitation in the current SimServer is the lack of visual representation of how components are connected to each other. A new SimServer solution is under development to address these issues, and it will also provide full support for the new MD307. It will separate the I/O and GUI modules from the MCU module, and as a result, SimServer will become more of a simulator suite than a single simulator.

1.2 Purpose

The purpose of this project is to design and implement a web-based user interface and I/O simulator for SimServer's MD307 microcontroller simulator.

By making it web-based, the application will be visually and functionally the same regardless of the operating system. It will also be possible to run it on a web server, which would further simplify distribution and maintainability by centralizing

bug fixes and updates. The I/O simulator should support the most important I/O peripherals that are used in the MOP course. To evaluate the simulator's functionality, programs used in the laboratory sessions will be executed on SimServer. These programs will be used to verify the interaction between the I/O simulator and the MCU simulator.

During this separation, the main goal of the design is to create a more intuitive and modern GUI. This will be achieved with a combination of identified user needs, having the user experience closely resemble the workspace in the physical laboratory sessions, improving intuitiveness by visually displaying the wires between I/O peripherals, and focusing on creating a simple and minimal design. This aspect will be evaluated by conducting feedback sessions with students with prior experience in the MOP course.

1.3 Scope

Since the MCU simulator is already provided, its development is outside the scope of the project. The I/O simulator was developed with the MOP course in mind, and therefore any use outside the course is not considered.

In terms of functionality, the project prioritized the implementation of basic I/O peripherals essential for laboratory exercises. These consisted of the Dual In-Line Package switch (DIP switch), bargraph, 7-segment display, and keypad. More complex peripherals such as the graphic display, ASCII display, and buzzer required considerations into precise timing behavior and were not included due to time constraints. Touchscreen optimization for mobile phones is excluded because of its limited relevance and additional complexity.

Worth noting is that a separate student group is working on backend development related to this simulator. However, integrating the final web-based application with their codebase is not within the scope of this project.

2
Theory

This chapter presents the theoretical background necessary to understand this project's implementations and design choices. It begins with an overview and explanation of programming languages and the Machine-Oriented Programming course, followed by descriptions of the two instruction set architectures (ISAs) employed by the current MD407 and the new MD307 MCUs. Details of both microcontrollers and their peripheral I/O units are also provided. Next, it provides insight into the value of simulators, why they are used, and specifically how the currently available version of SimServer works. The WebSocket communication protocol, which is essential to understand the communication between the I/O simulator and the MCU, is then described. The chapter ends with an explanation of the web development technologies used.

2.1 Programming Languages

A programming language is a set of keywords that are written according to a predefined syntax, often unique to that specific language. This allows the computer to execute the code either by compiling it to machine code prior to execution or interpreting it during execution. This section describes the languages relevant for the project.

2.1.1 JavaScript

JavaScript is a high-level interpreted programming language and one of the core technologies used in web development. It enables developers to create interactive and dynamic web content by manipulating the structure and behavior of web pages in real time. JavaScript is dynamically typed and supports several programming paradigms, including imperative, functional, and event-driven programming, which gives flexibility across various applications. JavaScript allows for dynamic updates of page content, and user input handling, without requiring a page reload. The language also includes built-in support for asynchronous programming, allowing responsive client-side applications [3].

2.1.2 TypeScript

TypeScript is an open source programming language developed and maintained by Microsoft. It is a syntactic superset of JavaScript, which means that all valid

JavaScript code is also valid TypeScript code. However, TypeScript introduces additional features, most notably optional static typing, which allows developers to declare variable types, that are checked at compile time. This feature helps to catch errors early in the development process and makes the code easier to understand and maintain [4].

2.1.3 The C Programming Language

C is a compact and efficient systems programming language that combines low-level access memory with structured programming constructs [5].

C supports basic data types including `int`, `char` and `float`, and more complex derived types such as arrays, structures, unions and pointers. Programs in C are composed of functions and its control structures `if`, `for loops` and `switch-cases`. One of C's most defining features is its close relationship to machine-level operations, giving programmers fine-grained control over memory and performance.

2.1.4 The C++ Programming Language

Unlike its predecessor C, the C++ programming language was designed to address the growing need for abstraction in large systems. C++ extends C with mechanisms for data abstraction, object-oriented design, and generic programming [6]. This combination allows developers to write more modular and maintainable code without sacrificing performance.

2.2 Machine-Oriented Programming

Machine-oriented programming emphasizes direct interaction with computer hardware, prioritizing control over system resources such as memory, registers, and instruction execution. It is distinct from high-level programming in that it avoids abstraction, enabling the programmer to work closer to the hardware to achieve optimal performance and predictability [7].

At its core, machine-oriented programming builds upon the principles of computer architecture, including instruction sets, memory hierarchies, and I/O mechanisms. The programmer is responsible for handling aspects such as memory management, stack operations, and register usage manually, often writing in assembly language or low-level C [8].

2.2.1 Course MCUs and their Instruction Set Architectures

Currently, there are two MCUs used in the MOP course. MD407 is the older of the two and will be used until a full transition to the newer MD307 is possible. The instruction set architecture (ISA), essentially the instructions which the CPU have available, differs between the two MCUs. MD307 uses RISC-V (Reduced Instruction Set Computer-Five) and MD407 uses ARM (Advanced RISC Machine).

Both of these architectures belong to the RISC-family of ISA, which strives to keep the instruction set as simple and small as possible. Simpler instructions can lead

to less power consumption and better hardware utilization, which is often favorable in embedded devices [9]. From an educational perspective, RISC architectures are often favored since their simplified instruction sets make it easier for learners to focus on fundamental concepts without being overwhelmed by the large number of instructions.

Even though both are considered RISC, one major difference between them is that RISC-V is open source, allowing anyone to use it without license fees [10], while ARM is proprietary [11]. The alternative to RISC is Complex Instruction Set Computer, which typically has more instructions, as well as having more complex instructions capable of performing multi-step instructions.

2.2.2 GPIO Ports and Pins

The MD307 MCU houses two 16-bit GPIO ports, port D and port E, each of which is split into two halves on the hardware and arranged as shown in Figure 2.1. The lower half contains pins 0-7, while the higher half contains pins 8-15. Each half also includes a voltage pin and a ground pin, which means that it consists of 10 pins in total, illustrated in Figure 2.2. In the rest of this thesis, the term *port* refers specifically to these 10-bit partial ports of the full 16-bit ports.

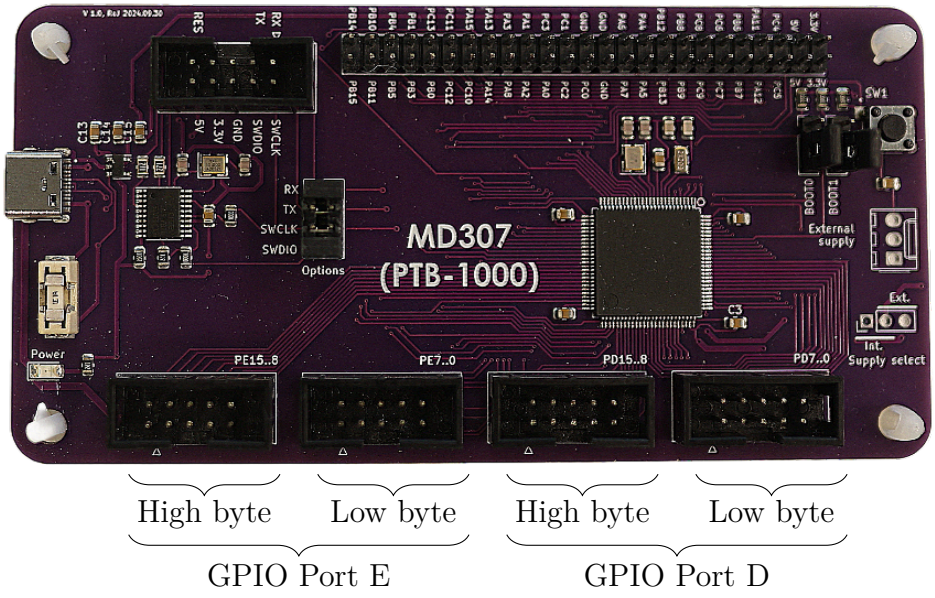


Figure 2.1: GPIO port layout on the MD307 MCU. GPIO ports D and E are each split into high and low bytes. Each pin corresponds to a bit in the byte, with two additional pins for voltage and ground.

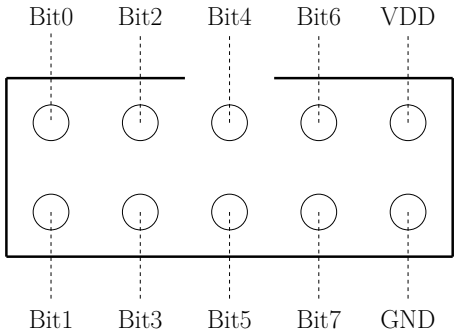


Figure 2.2: Illustration of the pin layout of a GPIO port. The Voltage Drain (VDD) pin provides power, while GND is the ground pin. The gap at the top ensures proper wire orientation.

2.2.3 Peripheral I/O Units

To connect a peripheral I/O unit to the MD307 MCU, a 10-pole flat cable is used to connect the peripheral port to one of the MCU ports, as shown in Figure 2.3. The available hardware peripherals are the bargraph, DIP switch, keypad and the 7-segment display.

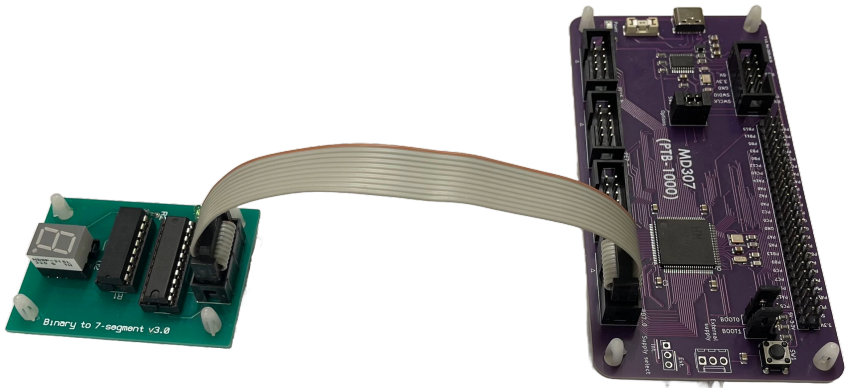


Figure 2.3: Hardware MD307 connected to a 7-Segment display, using a 10-pole flat cable.

The bargraph is a basic output peripheral that contains ten LED lighs, as shown in Figure 2.4. Two of these are unused, while the other eight are used to visualize the connected port state. Each LED lights up when the corresponding pin is powered.

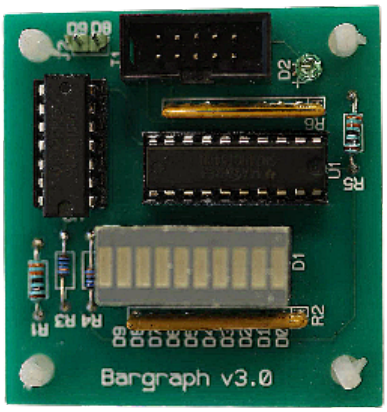


Figure 2.4: Hardware bargraph component used in the MOP course. The port at the top connects to the MCU, with each of the pins D0-D7 controlling its corresponding LED in the center. The LEDs labeled D8 and D9 are not functional with MOP course MCUs.

The DIP switch is an input peripheral that consists of eight switches. When a switch is on, the associated pin of the port is powered. The physical DIP switch component is shown in Figure 2.5.

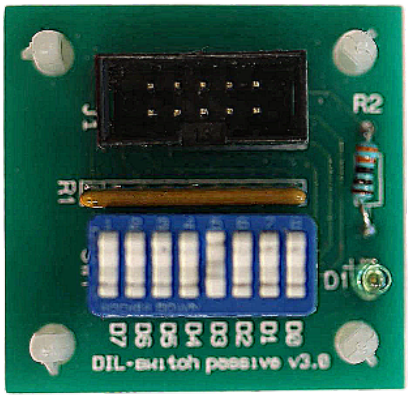


Figure 2.5: Hardware DIP switch component used in the MOP course. The eight central white toggle switches, labeled D0-D7, control the state of its corresponding port pin.

The keypad peripheral contains sixteen keys, ten of which are labeled 0 through 9, four are A through D, and the remaining two are octothorpe and asterisk, as shown in Figure 2.6. Unlike the switch, the keypad contains more keys than available pins. Because of this, the 16 keys are wired as a 4x4 matrix, which allows pressed keys to be identified through a scanning method. This method entails activating rows while reading the column states, with pressed keys completing the circuit between an activated row and the key's corresponding column.

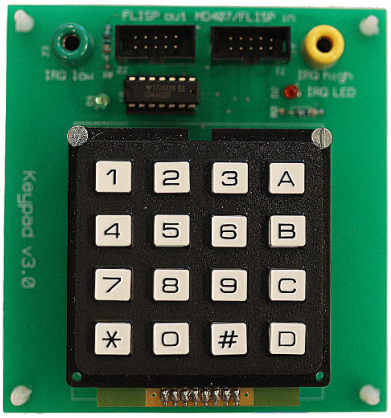


Figure 2.6: Hardware keypad component used in the MOP course, showing the 16 buttons labeled 0-9, A-D, * and #. Only one of the two ports at the top are used in the MOP course.

The 7-segment display uses the eight pins to drive seven oblong segments and a decimal point circle segment, as can be seen in Figure 2.7. Within the MOP course, it is often used to display input to the MCU from the keypad peripheral in the form of a hexadecimal number.

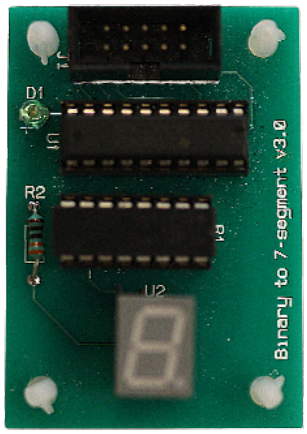


Figure 2.7: Hardware 7-segment display component used in the MOP course. Each of the seven segments and decimal point segment is driven by a corresponding port pin.

2.3 Simulators

Simulators are widely used tools in computer science and engineering, emulating the behavior of systems under virtual and controlled conditions. They allow for testing, debugging, and analysis without risk of damaging physical hardware. In general, a simulator replicates aspects of a system's structure and behavior, such as CPUs, memory subsystems, I/O buses or complete embedded platforms. This provides a cost effective and flexible approach to performance analysis and system validation [12].

A specific category of simulators are I/O simulators, which are built to model data transfer between a system's memory, processor, and external peripherals. These simulators are especially useful when designing or testing low-level systems, such as operating systems or firmware, where the behavior of disk, network, or peripheral interfaces must be analyzed [13].

In the context of machine-oriented programming, where software is written to interact closely with hardware, simulation becomes indispensable. Programs written in low-level languages such as C or assembly often require precise control of memory addresses, hardware registers, and I/O ports. Testing such programs directly on hardware can be impractical and risks damaging it, particularly during early development [14].

2.3.1 SimServer

SimServer is an integrated simulator written in C and C++, mainly aimed towards education in machine-oriented programming and systems programming. It is available on Windows, Mac and Linux computers and developed by Göteborgs Mikrovaror [15]. SimServer can simulate numerous target microcomputers that have been developed for education purposes, such as the previously mentioned MD407 and MD307 [16]. As seen in Figure 2.8, the SimServer program contains the MCU simulator together with the I/O simulator. The I/O simulator is responsible for the connection of peripheral units like the Bargraph, DIP Switch and Keypad. The user interface for SimServer, used to configure and visualize the peripheral units, is shown in Figure 2.9. The client and SimServer communicates using the GNU Debugger (GDB) remote server-protocol over the Transmission Control Protocol (TCP). This allows clients such as GDB and VSCode to run and debug programs.

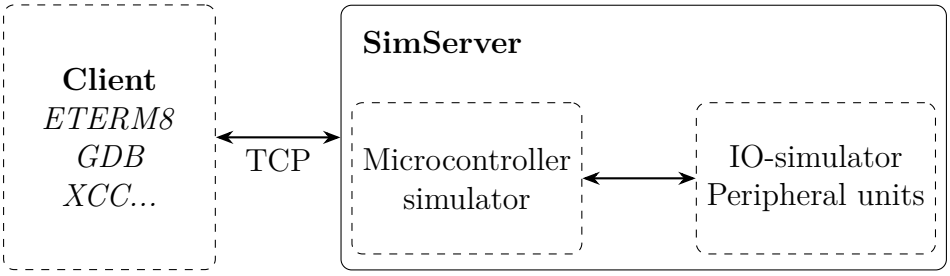


Figure 2.8: Shows the interaction between client and the currently available SimServer program consisting of two different kinds of simulators.

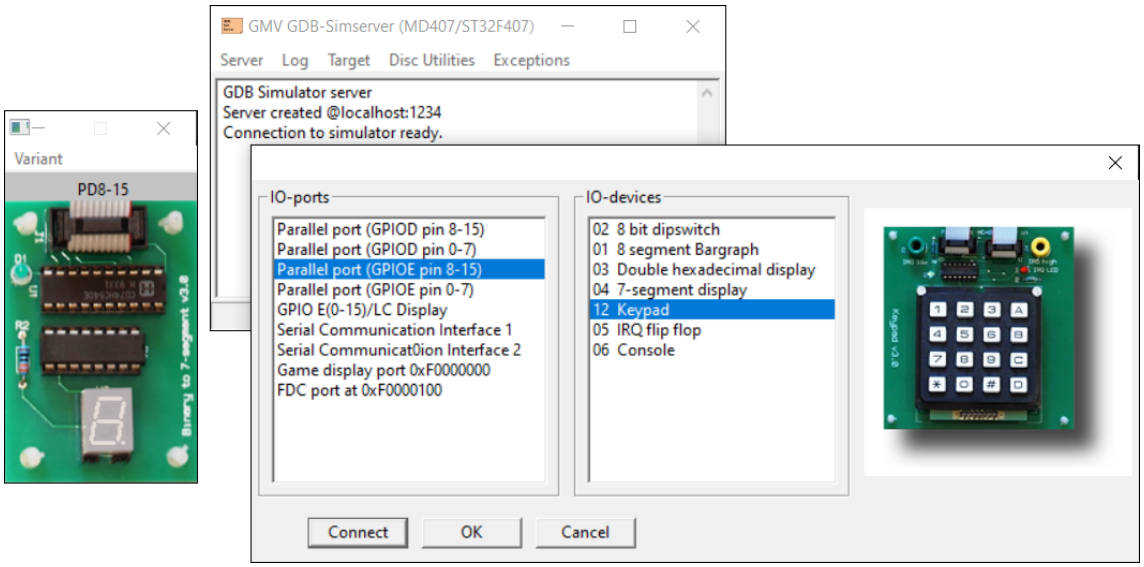


Figure 2.9: The current SimServer user interface. The left window shows a seven-segment display connected to the high byte of port D. The right window demonstrates how an I/O device is added and connected to a port. In this case, a keypad is selected for connection to the high byte of port E.

2.4 Web Development Technologies

The core languages used to build web applications are Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript [17]. HTML defines the meaning and structure of the application, using elements such as images, paragraphs, and buttons to organize the content [18]. This structure is represented in memory by the Document Object Model (DOM) [19]. CSS is a style sheet language that is used to describe how HTML elements should be displayed on the screen, for example the color, size, and margin of an element [20]. JavaScript, as described in Section 2.1.1, is a programming language used to implement dynamic features and interactivity on web pages, by using the DOM [21].

In modern web development, these languages are usually used in conjunction with frameworks and libraries. These can significantly influence the design, workflow, and performance of an application.

Frameworks are sets of tools and resources that developers can use to build and manage web applications more efficiently, with less errors, and with better performance and reliability. They may include, for example, application templates, libraries, or other modules with pre-built components or code snippets [22].

A library is a collection of pre-written and reusable components designed to solve common problems during development. By abstracting complex solutions behind simple interfaces, they save time, while also improving code quality and code readability. In the case of web development libraries, they also help ensure that the design remains consistent, especially when developing larger applications and when working in teams [23].

2.4.1 Svelte

Svelte is the main front-end framework used in this project. Instead of interpreting code in the browser, Svelte uses a compiler-based approach to transform declarative components, written in HTML, CSS, and JavaScript/TypeScript, into efficient imperative code that directly updates the DOM [24]. This approach retains the runtime performance and low memory consumption offered by plain Javascript, which results in smaller bundles, faster execution, and less boilerplate [25].

One of Svelte's defining features is its reactivity model. Svelte uses compiler analysis to automatically track which values are reactive and when updates should be applied [26]. Svelte *stores* are globally accessed reactive variables that hold values shared across components [27]. They enable components to automatically react to changes by subscribing to store updates. Svelte provides writable, readable and derived stores. Writable stores allow both reading and writing, while readable stores allow only reading. Derived stores derive their values from one or more other stores, and are often used to subscribe to part of a state or a transformed representation of it.

Svelte also has a feature called *runes* which offers a way of handling the reactivity and component states, bringing more clarity and explicitness to how reactive values and stores are managed [28]. The runes API replaces implicit behavior with a more functional and composable syntax.

An alternative to Svelte that was considered for this project is React. React is a popular and established JavaScript library for building and rendering user interfaces by combining small elements, such as buttons, text and images, into reusable nestable components [29].

2.4.2 Tailwind CSS

Tailwind is a utility-first CSS framework that enables styling directly in HTML. Tailwind defines utility classes, such as `text-sm` or `text-lg` for small and large typography respectively, that developers compose to create modern user interfaces without writing custom CSS [30]. While this approach does require developers to be familiar with Tailwind, it also avoids the rigidity pitfalls that component libraries often fall into. Additionally, Tailwind optimizes performance by only generating CSS for the styles used, ensuring that it produces the smallest CSS bundle possible [31].

2.4.3 Skeleton

Skeleton is a web design framework, built as an extension of Tailwind, that integrates seamlessly with Svelte and many other frameworks. It consists of three main features: its design system, extensions to Tailwind and a suite of pre-built components [32], [33].

The design system provides preset styles, a typography system, and utilities for themes and colors [33]. Skeleton extends Tailwind to include a variety of styles for basic components such as badges, buttons, and chips, which act as primitives used to create more complex interfaces. The main feature is the collection of functional components, such as sliders and popups, which are commonly used in web develop-

ment. Although Skeleton uses an opinionated approach, which means that it has some predefined solutions and can therefore result in some inflexibility, it also facilitates faster development and results in a more visually coherent and more accessible application [32].

2.4.4 Lucide

Lucide is an open source icon library that offers a wide variety of modern and customizable icons and symbols, recommended by Skeleton [34], [35]. It uses SVG compression and optimizes bundle sizes by ensuring that only the necessary size is used for icons in the final build.

2.4.5 WebSocket Protocol

The WebSocket protocol provides a way for a client, often a web browser, to establish bidirectional communication with a remote server [36]. When used in web browsers, WebSocket communication is handled by JavaScript code, which runs in a sandboxed environment that restricts access to system resources for security purposes. The server, or the remote host, must explicitly accept WebSocket connections.

The connection begins with a handshake over HTTP, after which it switches to a more efficient messaging format using TCP, enabling full-duplex communication between the client and server. The goal of the protocol is to allow applications, particularly those in the browser, to maintain ongoing, low-latency communication with servers without having to repeatedly open and close separate HTTP connections in both directions, which is common in older technologies.

2.4.6 JavaScript Object Notation (JSON)

JSON is a lightweight text-based data interchange format that is both readable for humans and parsable for machines [37]. It organizes data as key-value pairs and arrays, making it simple and flexible. Among various other use cases, JSON is widely used in web APIs. Many other alternatives to JSON exist, but none were considered since the provided SimServer backend already implements JSON messaging.

3
Methods

This chapter presents the application's design and implementation. As shown in Figure 3.1, the interface consists of two primary visual components, the *Event Log* for displaying messages, and the *I/O node editor*, which contains the I/O units and their connections. First, the design process for these components is described. Next, their technical implementation is detailed, along with the WebSocket and internal event system that handles communication with SimServer. The chapter concludes with the evaluation methodology, explaining the user tests performed to evaluate the final result.

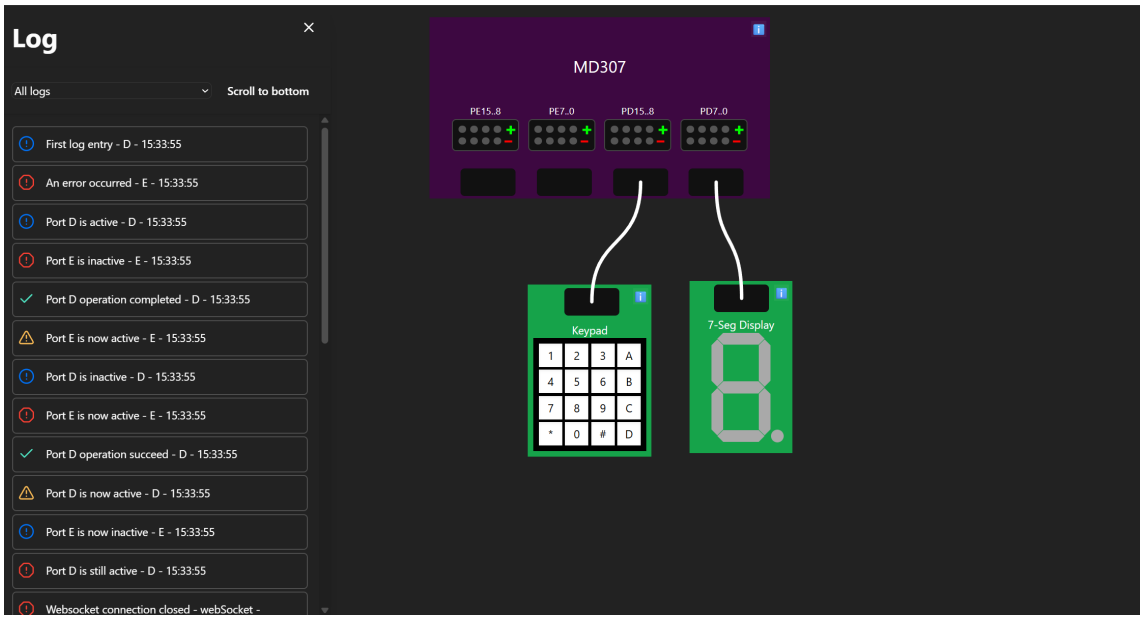


Figure 3.1: Overview of the application interface. The Event Log is open on the left, and the right shows the I/O node editor with a keypad and 7-segment display connected to the MCU.

3.1 User Interface Design

This section details the choices made for the design of the MCU and its peripheral I/O units, *I/O node editor*, and *Event Log*, with a focus on creating an intuitive and visually appealing interface. Additionally, it explains how early user interviews were conducted, and how they directly affected these choices.

3.1.1 Identified User Needs

To identify user needs and possible areas of improvement for the current simulator, interviews were conducted with students who had experience with the MOP course. There were four interviews in total and the participants consisted of two older students who took the course last year, one current teacher assistant in the MOP course, and one student who had just finished the MOP course. The interviews were recorded and transcribed and the questions can be read in appendix A.

The interviews followed a semi-structured approach, combining a set of predefined questions with the flexibility to ask follow-up questions and explore relevant topics in more depth [38]. The purpose of the interviews was to analyze students experiences with the current simulator, understand the challenges they encountered, and identify their expectations for the new simulator. This information was then used to guide design decisions and highlight potential usability issues. One of the main problems identified, was that the current simulator is difficult to use because of its non-intuitive design. Some interviewees felt that it was unnecessarily difficult to navigate the numerous menus, which, as a result, made adding components overly complicated. In response, emphasis was placed on prioritizing visibility and a cleaner and clearer user interface.

3.1.2 MCU and Peripheral I/O Units

The current SimServer interface represents the components as separate windows with interactive and dynamic photos of the physical hardware components. This type of design, where the digital components closely resemble their physical counterparts, is called a *skeuomorphic* design. Skeuomorphism can aid in making a visual connection between the physical and digital versions of the components, but it can also become unnecessarily cluttered [39]. This project chose a more minimalistic design approach for the I/O units, while still maintaining key visual features of the physical counterpart, as show in Figure 3.2 and Figure 3.3.

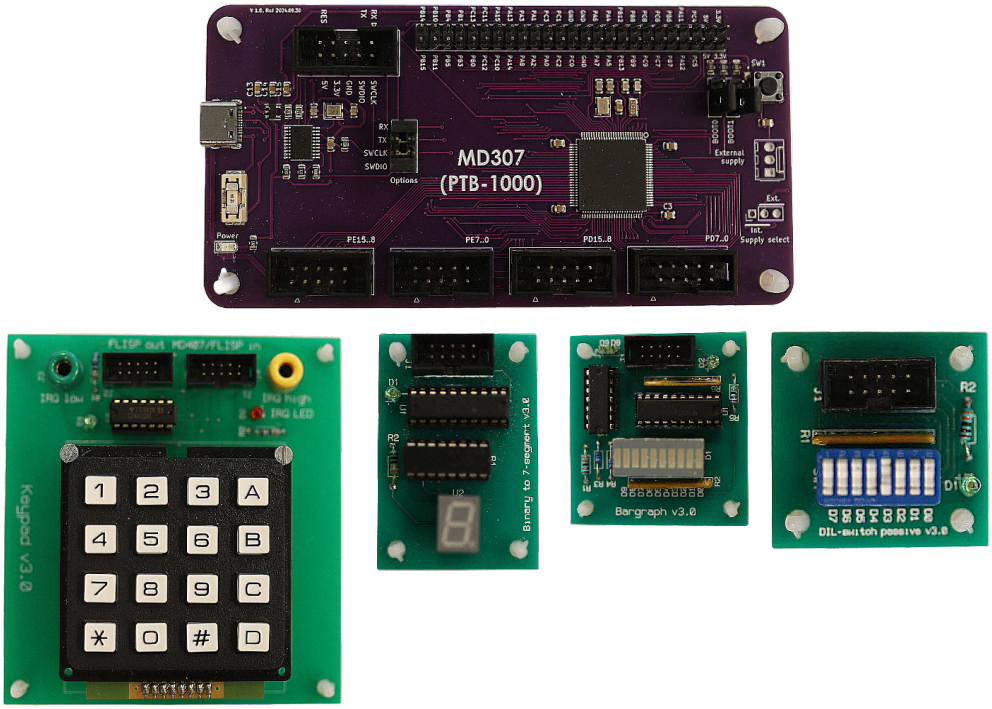


Figure 3.2: Shows the physical components used in the course with relative sizes.

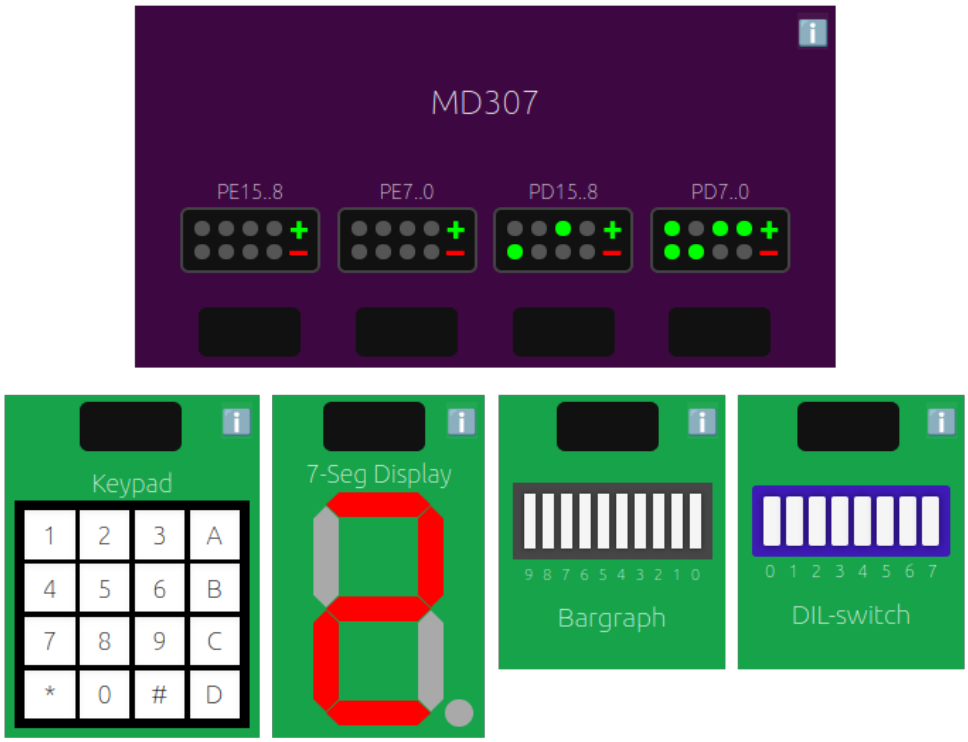


Figure 3.3: Shows the minimal design of the components in the new simulator interface.

3.1.3 I/O Node Editor

The main part of the application is a node editor that provides a visual interface to arrange components and configure the I/O setup. It is designed as a direct replacement for the abstract menu-based interface currently used in the MOP course. Instead of navigating nested menus, the node editor mirrors the workflow of the physical lab environment, allowing students to arrange components and connect them as they would with real hardware.

The core elements of the editor are the MCU, peripheral components and their connections. The MCU and peripherals (e.g. switches, bargraphs, keypads) are represented as draggable nodes that can be freely positioned within the editor's workspace, much like how students would arrange physical components during lab sessions.

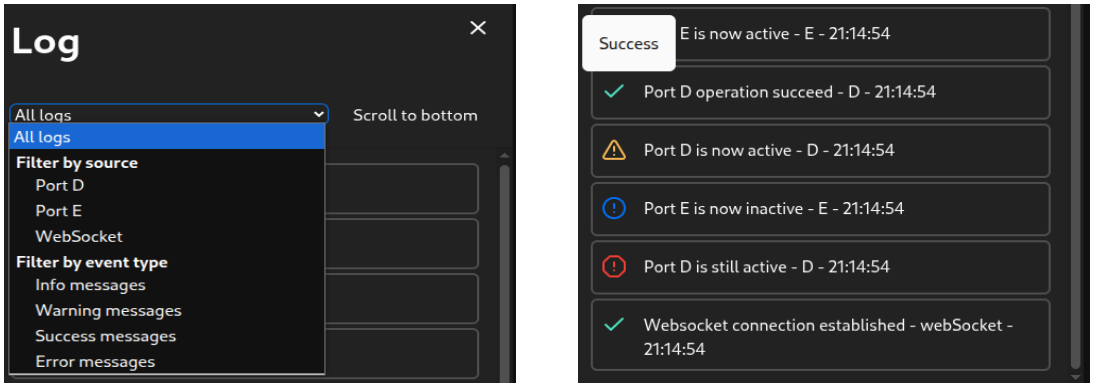
Connections between component ports are visualized as smooth wire-like curves, approximating the look of physical wiring used in laboratory sessions. These curves dynamically adjust to the position of the connected node ports to maintain a clear and intuitive representation for the connections between them.

3.1.4 Event Log

The event log is another part of the application, designed as a drawer HTML element that can be toggled on demand. It was designed to offer an informative and intuitive experience. Illustrated in Figure 3.4a are the filtering capabilities, where a drop-

down menu allows filtering by either event type or event source. The design of the log entries is shown in Figure 3.4b and is presented as well-formatted strings to improve readability. Each entry is also supported by an icon and an accompanying tooltip, visualizing the type of event that triggered it.

A button to instantly scroll to the latest entry was added, along with an automatic scrolling feature that activates when the log is already at the bottom. These additions were made to enhance the user experience by removing the need for manual scrolling when new entries are added.



(a) **(b)**

Figure 3.4: Shows an overview of the event log UI.
(a) shows the filtering drop-down menu and the scroll to bottom button. (b) shows the log entries and its associated event type with an icon and tooltip.

3.1.5 Themes

To aid in the availability for color-blind users, or simply to let users choose a theme that they enjoy, a theme system was implemented. This also adheres to the *Web Content Accessibility Guidelines* technique G174, that states that if a webpage does not have sufficient contrast ratio, there should be an option to change the page to an alternative version where the contrast is sufficient [40]. By allowing different themes, the application aims to provide better contrast ratio for color-blind users that might face issues with the colorful components. Just like the event log, users can press a button to display the theme selection, where they have the option to choose between five different pre-made color themes.

This theme system was made possible because most of the components make use of classes already defined by *Tailwind*. Some of these classes are also tied to specific color-variables defined in CSS files. Tailwind has one of these files by default, but it is easy to add more files with different color schemes. To allow users to change the CSS file used, a *ThemeSelector* was implemented, which simply switches out the CSS file based on the users input. A way to make sure that the selected theme is persistent even after reloading the webpage, was to make use of the *localStorage*. This way, when a new theme is selected, it is saved into *localStorage* which will be automatically used the next time the page is loaded.

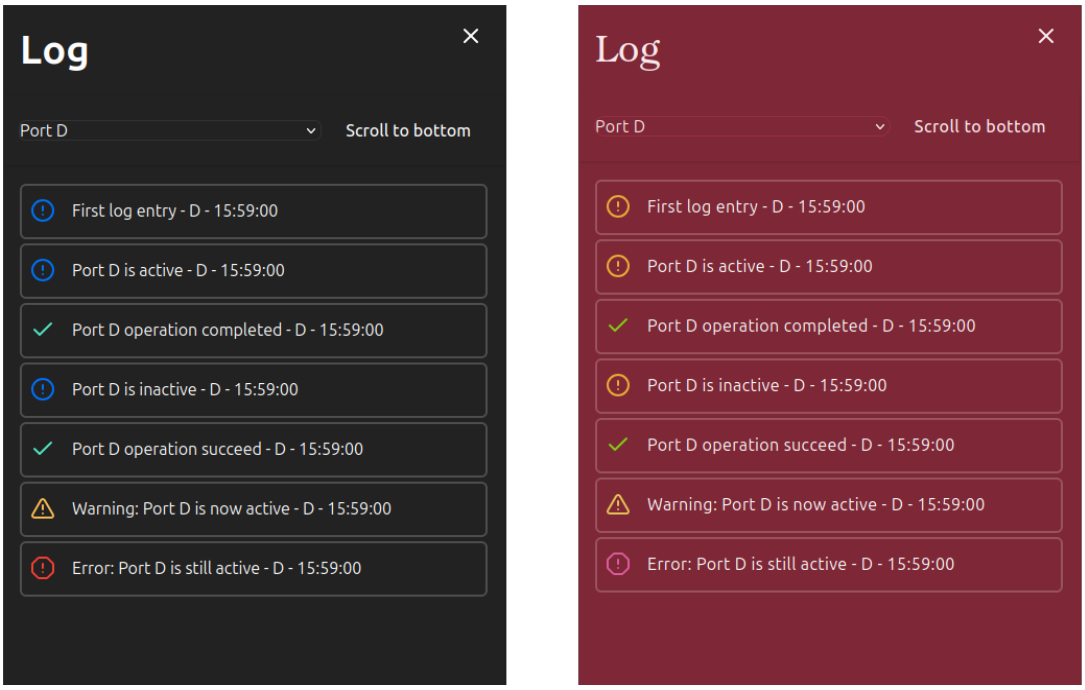


Figure 3.5: Shows the difference of two themes on the event log. (Cerberus left, Wintry right)

3.2 Implementation

This section details the implementation of the main components of the application and their interactions. The implementation of the application depends on several technologies. Svelte was chosen for its intuitive syntax and performant compiler-based approach, as detailed in Section 2.4.1. Its reactivity features were extensively utilized in most parts of the application. The selection of TypeScript provided static type checking that would result in more readable code with fewer bugs, as discussed in Section 2.1.2. For visual consistency and design, Skeleton was chosen for its component library, design system, and established integration with Svelte. Lucide provided an extensive library of icons and was recommended by Svelte.

The application receives WebSocket messages from SimServer, which are parsed and propagated using an internal event bus to the *I/O node editor* and *Event Log*. The I/O node editor provides the interface for working with I/O units and their connections. The Event Log displays messages from both the WebSocket and the internal application content. The relationships between these systems are shown in Figure 3.6, with each system being discussed in further detail in its separate section.

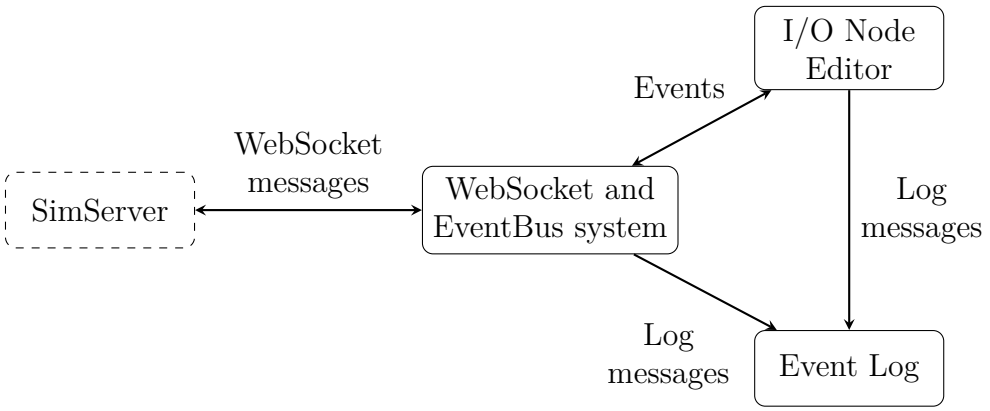


Figure 3.6: Application architecture overview. Data flows through the WebSocket connection with the external SimServer, and is distributed to internal application content using the EventBus. The Event Log receives log messages from both the WebSocket and I/O Node Editor components.

3.2.1 SimServer Communication Protocol

The message protocol used for the communication between the application and SimServer utilizes a JSON format containing a `type` field that identifies the type of event (e.g. 2 for GPIO input), and a `data` payload containing a type-specific object, according to Figure 3.7. Types 0 and 1 are defined in SimServer as `EMPTY` and `UNKNOWN` respectively, but are not used by this application.

Types 2 and 3 are for GPIO updates and consist of the port and the new pin state. Type 2 also includes a 64-bit timestamp, indicating what cycle SimServer was on when the message was sent. Because JSON lacks native support for 64-bit integers, two 32-bit integer fields (`cycles_low` and `cycles_high`) are used instead, which are then reconstructed using JavaScript's `BigInt`. This is necessary for simulating components that depend on timing, such as buzzers or LCD displays.

Types 4 and 5 are currently only used by the keypad and sent to SimServer. These are used to connect or disconnect two pins directly in SimServer as a workaround to the inherent latency of the WebSocket. This is referred to as a *bridge* in the implementation.

```
{
  "type": "2",
  "data": {
    "port": "D" | "E",
    "pins": [
      { "value": boolean },
      ...
    ],
    "cycles_low":
        <low 32 bits of
        cycle count>,
    "cycles_high":
        <high 32 bits of
        cycle count>
  }
}
```

Type 2: GPIO from SimServer

```
{
  "type": "3",
  "data": {
    "port": "D" | "E",
    "pins": [
      { "value": boolean },
      ...
    ]
  }
}
```

Type 3: GPIO to SimServer

```
{
  "type": "4",
  "data": {
    "port": "D" | "E",
    "fromPin": <the source
    pin for the bridge>,
    "toPin": <the target pin
    for the bridge>
  }
}
```

Type 4: Add bridge

```
{
  "type": "5",
  "data": {
    "port": "D" | "E",
    "fromPin": <the source
    pin for the bridge>,
    "toPin": <the target pin
    for the bridge>
  }
}
```

Type 5: Remove bridge

Figure 3.7: JSON schemas for message types sent between the application and SimServer.

3.2.2 WebSocket and Internal Event System

The WebSocket system consists of three main components: the `WebSocketProvider`, the `EventBus` and the `WebSocketManager`. The `WebSocketProvider` initializes both the `EventBus` and the `WebSocketManager`. This structure is illustrated in Figure 3.8.

The `WebSocketManager` contains the core functionality of the WebSocket communication system. It establishes and maintains the connection to SimServer, parses incoming WebSocket messages into events and emits them through the `EventBus`, while also listening for events to serialize and send to SimServer. The manager validates messages and handles errors, rejecting invalid JSON or unsupported message types before they reach application content.

The `EventBus` acts as the primary internal communication layer within the application. It implements a publish-subscribe pattern that abstracts the WebSocket layer from the rest of the application. Using Svelte's context API, the `WebSocket-`

`Provider` makes the event bus available throughout the application. This structure allows any part of the application, i.e. the MCU and Log, to access the bus to emit or listen to events, without having to handle raw WebSocket messages.

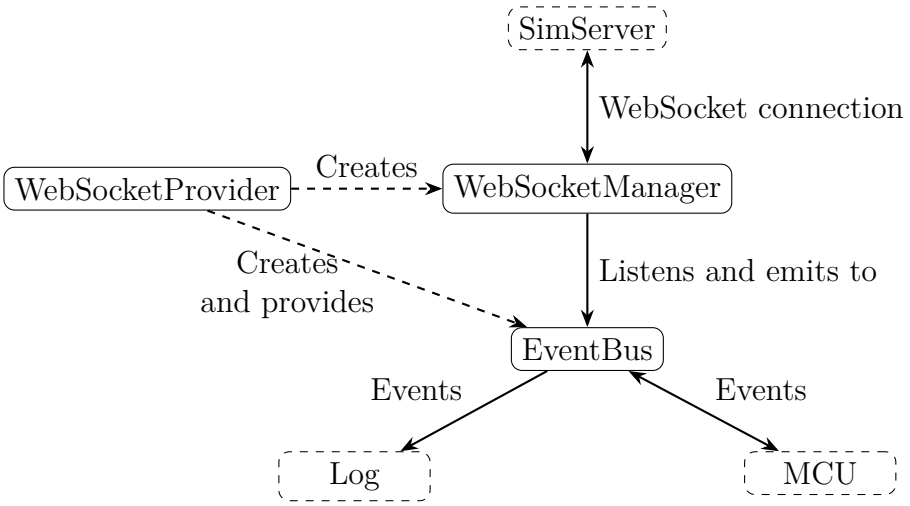


Figure 3.8: Interaction between SimServer, WebSocketManager and application content via the EventBus. Solid boxes are internal components of the event system. Dashed boxes are external components.

3.2.3 Event Log

To handle log data efficiently, the implementation of the Event Log uses Svelte's built-in reactivity model, and specifically a reactive store, described in Section 2.4.1. An overview of the log implementation is shown in Fig 3.9. The `logStore` holds an array of `Event` objects with the specified fields: `eventType`, `source`, `message`, and `time`. The `Events` are added to the store using the `addLog` method. The `logStore` retains only the last 100 events, and allows updates to propagate to the UI.

Filtering was implemented through a separate `activeFilter` store, containing the currently selected filter criteria, filtering on either `eventType` or `source`. `eventType` can take any of the values `'Info'`, `'Warning'`, `'Succeed'`, or `'Error'`, and `source` can be either WebSocket or one of the GPIO ports D or E. A derived store, `filteredLogs`, was used to derive a subset of `logStore` based on `activeFilter`.

Lastly, to improve user experience by making the log both intuitive and informative, a filter selection menu was implemented. The filter selection menu uses an `HTML<select>` element. That input updates `activeFilter`, which dynamically derives a subset of `logStore` through the `filteredLogs` store. Depending on `eventType`, each entry is rendered with a colored icon from the Lucide icon set. Tooltips were implemented using a template from Skeleton, enabling additional information when hovering over the icons. The auto-scroll mechanism was implemented by introducing logic that checks if the user is at the bottom of the scroll container. If so, a trigger monitors changes in `filteredLogs` and activates the scroll logic.

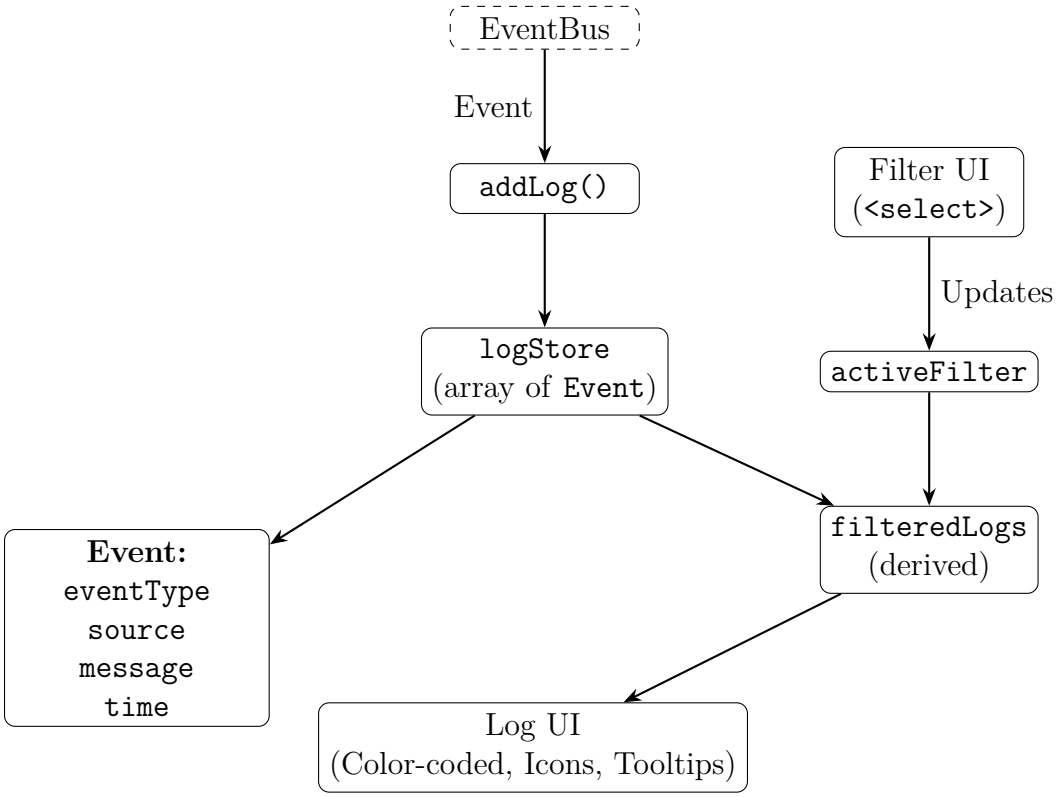


Figure 3.9: Overview of the log system showing event flow, structure, and filtering. Solid boxes are internal components of the log system. Dashed boxes are external components.

3.2.4 I/O Node Editor

The I/O node editor, referred to as the `Sandbox` in the codebase, is the workspace used for adding, arranging and connecting peripherals. Visual rendering and user input is handled by the `Sandbox` Svelte component, while the underlying logic for node and connection management is managed by the `SandboxManager` class and its two submanagers.

The logic is structured around three core managers, each responsible for a different part of the editor logic. The `NodeManager` handles node creation, selection, and movement, ensuring that multi-selection and drag movements are applied correctly to relevant nodes. The `ConnectionManager` is responsible for validating, adding and removing connections between ports. The `SandboxManager` receives input events from nodes and ports, maintains a state for drag interactions for coordination, as illustrated in Figure 3.10, and delegates actions to the appropriate submanager.

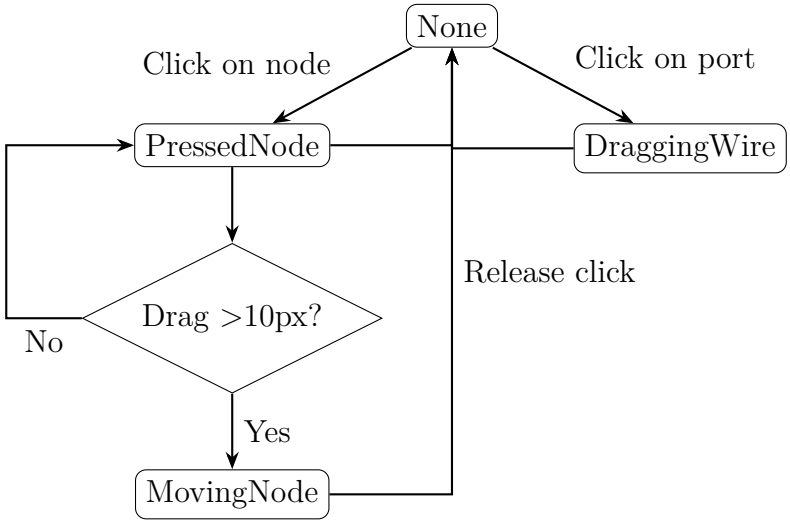


Figure 3.10: State machine diagram for drag interactions, illustrating transitions between different states when interacting with nodes and wires in the interface.

The `Node` Svelte component serves as an interactive container for peripheral components. It receives a `NodeData` instance as a prop, which maintains the node's state. This state includes positional coordinates, peripheral type and selection status. It dynamically updates along with these attributes, applying CSS transforms for positioning and setting a white border color when selected, and black otherwise.

It captures mouse events, `onmouseup` and `onmousedown`, and delegates them to the `SandboxManager` using callback props. These callbacks contain two variables, the node's unique id, and a boolean flag indicating whether the Ctrl key was pressed during the interaction. This allows the `SandboxManager` to update the position and selection state of relevant nodes. The Ctrl key allows the user to select and move multiple nodes.

The `NodeData`'s `type` property determines which peripheral component (bargraph, switch, etc.) is injected into the `Node` container. Each peripheral component encapsulates and handles its own rendering, functional logic and port configuration.

3.2.5 Ports and Connections

The port and connection systems consist of three main parts. The `PortComponent` is responsible for visual representation and delegating interaction events. Upon initialization, it creates an instance of the `PortData` class which contains the port's state, event handling logic, and utility functions. Port states are coordinated and modified by a central `ConnectionManager` class.

Additionally, a `Pin` Svelte component is used to visually represent individual pin states. The port component renders 10 of these in a 2x5 grid, with 8 pins used by SimServer and peripherals. The remaining two serve as static visual indicators for the power supply pin (always on) and the ground pin (always off).

Through its props, the port component enables peripherals to register a callback function for three different event types using the PortData class. These events are `onChange` for pin state change events, `onConnect` for connection events, and `onDis-`

`connect` for disconnection events. This allows peripherals to respond dynamically to port state changes.

The `onChange` callback provides peripherals with updates to the port's pin state and a timestamp in cycles for when that change occurred. For example, the 7-segment display uses this to refresh its numerical display based on the latest pin state. The `onConnect` callback is called when a new connection is created, and provides a way for peripherals to initialize with the current pin state. Finally, the `onDisconnect` callback enables peripherals to reset their state when disconnected. This callback is used by the bargraph peripheral to turn off all LEDs when invoked, and by the keypad to remove all current bridges.

Connections are established through interaction between the ports, the `SandboxManager` and the `ConnectionManager`. When a user initiates a connection by dragging a wire from a port, the `SandboxManager` first updates the drag state to `DraggingWire` and then forwards the event to the `ConnectionManager`.

When the `ConnectionManager` receives the event, there are two cases. If the port has no existing connection, it stores the port as the source port for the drag action. For an already connected port, it instead designates the connected port as the source and removes the current connection. While the Sandbox's drag state is set to `DraggingWire`, it renders a `Wire` component with one end fixed to this source port and the other end tracking the mouse position.

To complete the connection, the user must release the drag on a valid target port, or the action is canceled and the drag state is reset to `None`. A valid connection can only be made between an MCU port and a peripheral port. The `ConnectionManager` enforces this by requiring an MCU port (`isMcu: true`) to only connect to a peripheral port (`isMcu: false`), or vice versa. This inherently also prevents other invalid connections, such as a port being connected to itself or to another port on the same component.

When a valid connection is established, the `ConnectionManager` updates both connected `PortData` instances. First, both store a reference to the other port. This allows both ports to be property cleaned up when disconnecting. Next, the peripheral port is configured. A reference to the MCU port's `PortStateManager` is set and the `onConnect` callback is triggered with its current pin states. It also registers the `PortStateManager`'s `onChange` listener to receive future updates. The MCU port remains unchanged, since it is already configured to receive state updates, explained further in Section 3.2.7. When either connected port is disconnected, the `PortConnectionManager` resets both ports by clearing the reference to the other. For the peripheral port, the `PortStateManager` property is reset, and its change listener is unregistered.

3.2.6 Wire Rendering

The `Wire` Svelte component implements dynamic visual connections between nodes, and serve as the visual representations for connections between peripherals and the MCU. The `Wire` component heavily utilizes the `$derived` Svelte rune to dynamically render a cubic Bézier curve using the HTML `svg` and `path` elements. As seen in Figure 3.11, a cubic Bézier curve uses a set of four control points, $\mathbf{P}_0$ through $\mathbf{P}_3$,

to define a continuous smooth curve according to a parametric equation [41].

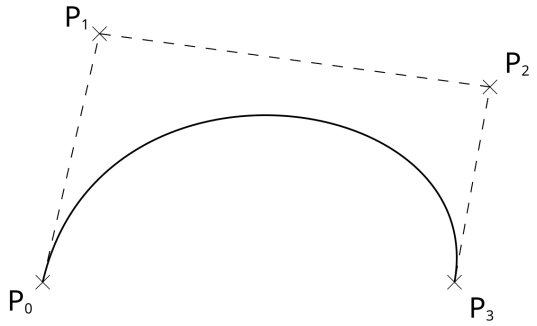


Figure 3.11: Cubic Bézier curve with 4 control points. [42], Public Domain

In the Wire component implementation, the endpoints $\mathbf{P}_0$ and $\mathbf{P}_3$ are anchored to each of the ports' positions. The remaining control points, $\mathbf{P_1}$ and $\mathbf{P}_2$, are derived from these anchors through a directional offset, explained by:

$$\mathbf{P}_1 = \mathbf{P}_0 + \mathbf{D}_a \cdot \ell$$

$$\mathbf{P}_2 = \mathbf{P}_3 + \mathbf{D}_b \cdot \ell$$

where $\mathbf{D}_a$ and $\mathbf{D}_b$ are unit vectors determined by the `WireDirection` of each port, and ℓ is the dynamic offset length $\ell = \min(d \cdot 0.5, 250)$, with d being the dynamic Euclidean distance between the connected ports. This dynamic scaling results in a larger curvature for more distant connections, making them more readable. The offset is scaled by 0.5, to make shorter connections smoother, and bounded by a 250px maximum to avoid overly exaggerated curves.

3.2.7 MCU

The MCU component provides the visual representation of the MCU in the editor and contains four ports. The configuration of these ports differs from peripheral ports in that they are initialized with the `isMcu` flag and a predefined port through the `initialPort` prop. This means that the pin visuals of these ports are updated regardless of active connections and that peripheral ports can be connected to them.

3.2.8 Bargraph

The bargraph component is an output device that visually represents the state of its port through eight LEDs. This is implemented using a boolean array where each element corresponds to the on/off state of an individual LED, which reactively update the visual displays. It contains a single port, configured with `onChange`, `onConnect` and `onDisconnect` callbacks. The `onChange` and `onConnect` functions propagate the new pin states to the internal state to update the visual displays, while the `onDisconnect` function resets the display by setting all array values to false.

3.2.9 7-Segment Display

The implementation of the 7-segment display functions similarly to the bargraph component as an output peripheral, maintaining an internal boolean array where the elements correspond to the 7-segments and the decimal point. It implements identical callbacks through its single port, meaning that `onChange` and `onConnect` update the display state, while `onDisconnect` resets all segments. The difference is that the 7-segment display visualizes each pin as a segment. This is implemented using HTML's `svg` element with a `polygon` element for each of the segments and a `circle` element for the decimal point.

3.2.10 Dual In-Line Package Switch

The DIP switch component is an input peripheral, meaning that it transmits state changes to the connected port. The implementation consists of eight toggle switches, each corresponding to one pin in the port. When a user interacts with a button, the associated pin is toggled and the new state is propagated using the port's `sendGpio` method. The DIP switch contains a single port component, but does not utilize any of the callbacks since its state does not depend on external input.

3.2.11 Keypad

The keypad peripheral contains sixteen keys arranged in a 4x4 matrix. With physical hardware, the MCU can identify pressed keys through a scanning method, in which it activates rows while reading the column states, with pressed buttons completing the circuit between an activated row and its corresponding column.

This approach cannot be directly implemented in the simulation due to the limitations of WebSocket communication. The inherent latency in message passing between the front-end application and the SimServer back-end causes desynchronization. This means that by the time a column state update arrives from the front-end, the back-end simulation has typically already progressed to scanning subsequent rows, resulting in incorrect key identification.

To solve this desynchronization problem, the keypad implements direct pin bridging internally within the SimServer back-end. When a button is pressed, an `ADD_BRIDGE` message specifying the port, source pin (row) and target pin (column), is sent to SimServer. When the button is released, a corresponding `REMOVE_BRIDGE` message is sent to clear the bridge. The bridge allows SimServer to read the correct column pin state directly, based on the bridged pin state, without having to take communication delays into account. Although WebSocket latency still exists for bridge updates, this delay does not affect users in a perceptible way.

3.3 Code Quality

Ensuring code quality was an important task in the project. First, it was crucial for functionality purposes, as the application was intended to be used in the MOP course eventually. Second, it helped maintain code consistency, detect errors, facilitate code

reviews, and save time due to integration directly into Visual Studio Code in parallel with the continuous integration pipeline (CI pipeline). This was especially helpful in a group setting or for future project takeovers.

The CI pipeline combined a linting and formatting setup, using ESLint to check code quality and Prettier to enforce consistent code style. They were chosen for their maturity and seamless Svelte plugin support. This ensured that the entire project followed a consistent coding standard based on recommended styles for Svelte [43]–[45].

The setup was combined with a Git pre-commit hook solution called Husky, to make the whole process automatic [46]. Husky ensured that ESLint and Prettier ran prior to each commit, accepting the commit only if it conformed to the defined code standards. This was important in order to make the setup as seamless as possible.

The third step in the code quality process was peer reviewing, where each one of the 36 pull request on GitHub was audited by other members of the project. Following this pipeline increased code quality and reduced the occurrence of bugs.

In addition to peer reviews, a lot of the code was written during pair programming. This allowed two or more group members to code together in real time, which was especially useful when working on the same task or issue. Pair programming also contributed to better code quality by enabling continuous feedback and early detection of potential issues and bugs.

3.4 Evaluation Methodology

To evaluate the final product, user tests were conducted with students who had previously completed the MOP course. Since these students had experience with the current simulator used in the course, they were able to provide relevant comparisons and feedback. There was a total of 12 students who participated in the evaluation and none had participated in the early interviews, which means no one had seen the product earlier. Each participant was asked to test the application and then fill out a questionnaire aimed at assessing various aspects of the user experience, with the option to write comments about each rating. The questionnaire consisted of 10 statements and 10 comment sections. The participants rated how much they agree with a statement using a number between 1 and 5, where 1 was considered "Do not agree at all" and 5 was "Completely agree". By allowing comments after each statement, the students were given the opportunity to share their opinion without the possible discomfort of voicing criticism in front of the interviewer, resulting in more honest feedback.

The purpose was primarily to assess user experience, interface design, and the application's educational value. Each statement was therefore constructed with the intention of testing at least one of these requirements. User experience was tested by trying to understand how the students interact with the application and to identify potential flaws that prevent the students from using it as intended. To evaluate the interface design, opinions on the aesthetic of the application was needed. This included thoughts on colors, fonts used, but also the design of the peripherals. Finally, to evaluate if the goals of the thesis have been achieved, it was necessary to know if the new simulator had any potential educational value. This was done by

making the students consider its possible use during the MOP course and laboratory sessions, as well as comparing it to the currently used SimServer.

4
Results

A total of 12 students participated in the evaluation. The results are presented in subsections, and diagrams are included to illustrate the students' responses to the questionnaire statements. Overall, the feedback received from the interviews was positive, indicating that most of the participants found the I/O simulator useful. One of the most appreciated features was the log, with relatively high ratings. Another appreciated aspect was the visual appeal of the user interface. While most comments regarding the design were positive, several participants provided suggestions for improvements. Some noted that the I/O simulator would be more effective with additional support features. Despite some feedback to consider for future development, most of the participants would still choose this new simulator over the current one because of its visual appearance and usability, provided that all features are fully functional.

4.1 User Experience

To assess the general user experience, participants were asked to evaluate the application based on their interactions with it. The questions in this section focused on whether users understood how to perform tasks, whether the information was easy to understand, and whether the tool was perceived as helpful.

Question: I understand how to perform different tasks in the application

As shown in Figure 4.1, participant ratings are generally positive, with 11/12 responses being 3 or higher. This suggests most participants generally had a good understanding of how to use the application. However, many also commented that it was confusing at the start. One respondent commented that some form of introduction or tutorial would have helped, especially since it was unclear whether they could drag wires to connect components.

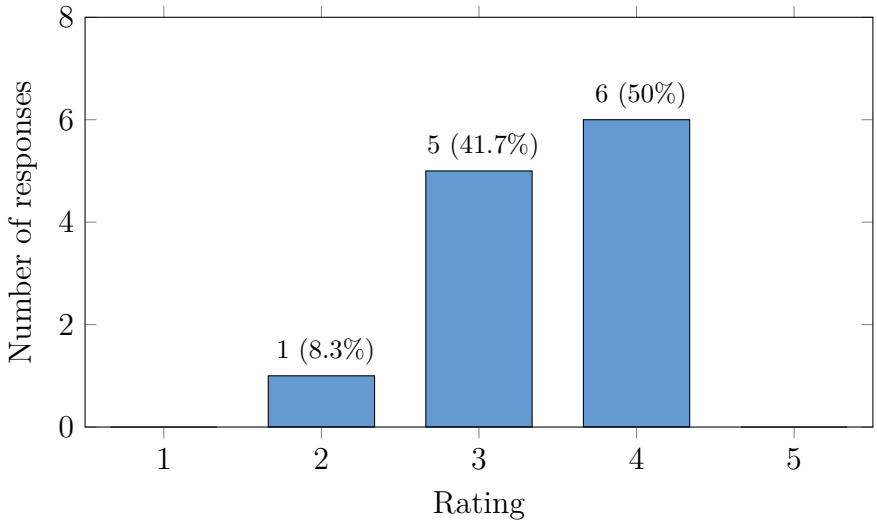


Figure 4.1: A graph depicting the number of responses and the distribution of ratings for the statement "I understand how to perform different tasks in the application".

Question: Necessary information is presented in a way that is easy to understand

As shown in Figure 4.2, all 12 participant ratings are 3 or higher, indicating a general understanding of the information presented in the application. A recurring comment on this question was that the information presented was helpful, but not sufficient, and that more explanation was needed to understand the full picture. Several participants also noted that a tutorial or some form of guidance would have been very helpful to get started.

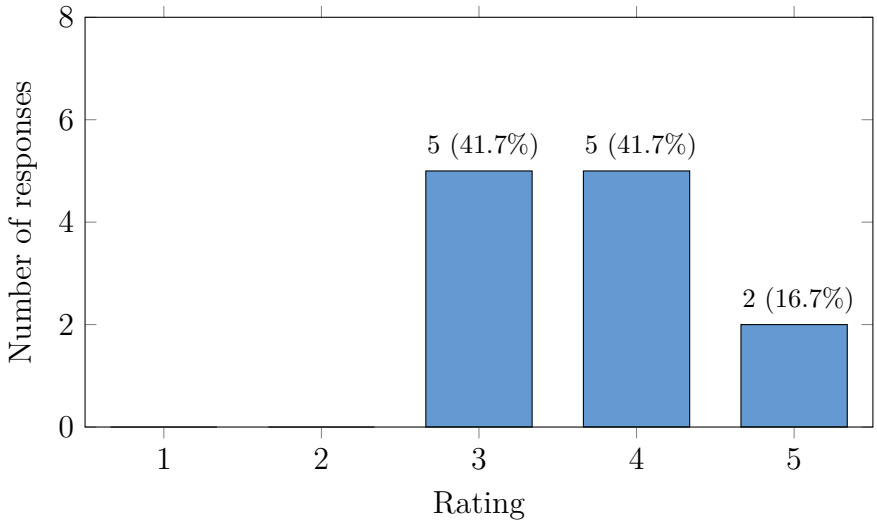


Figure 4.2: A graph depicting the number of responses and the distribution of ratings for the statement "Necessary information is presented in a way that is easy to understand".

Question: I think the messages in the log would be helpful

As shown in Figure 4.3, all participant ratings to this statement are 3 or higher, with half (6/12) of the ratings being a 5. Most participants who chose to comment on this question expressed that the information was generally easy to find and understand. The tooltips on the I/O peripherals were especially appreciated. One highly appreciated aspect of the log was the ability to filter by different types of messages. Some users expressed a desire for additional types of log messages and one comment mentioned that the usability of the log will depend on how it will work together with VSCode.

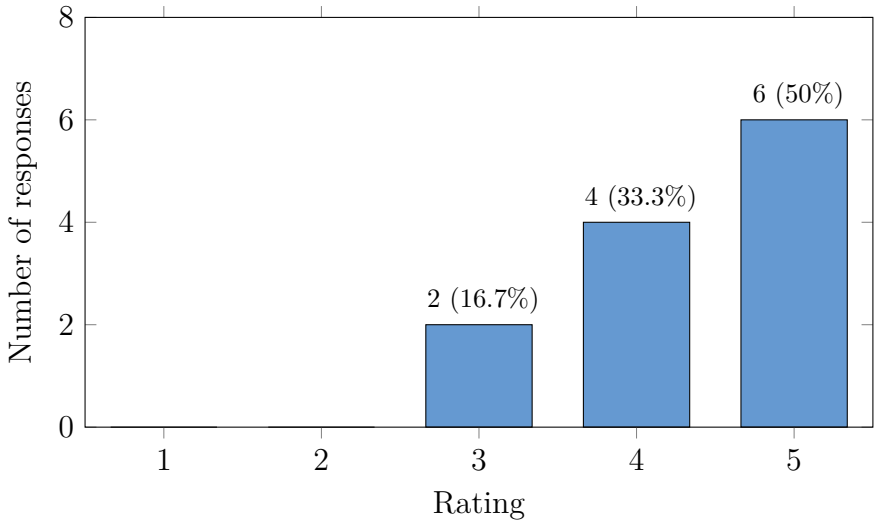


Figure 4.3: A graph depicting the number of responses and the distribution of ratings for the statement "I think the messages in the log would be helpful".

4.2 Interface Design

This section will take a closer look at the questionnaire statements and comments covering the interface design of the final application. It will specifically assess whether participants are satisfied with the aesthetic aspects such as colors, fonts, and the design of the peripherals.

Question: I think the user interface is visually appealing (colors, fonts etc.)

Figure 4.4 indicates the design of the interface was well received, with nearly all participants (11/12) rating it 4 or 5. Some participants thought that the new I/O simulator looked a lot better than the current simulator used in the course. Some stated that the application needs more work, and suggested improvements such as making the components more 3-dimensional and less flat using shadows, changing the color of the MCU to one that is easier to see, as well as improving the look of the ports.

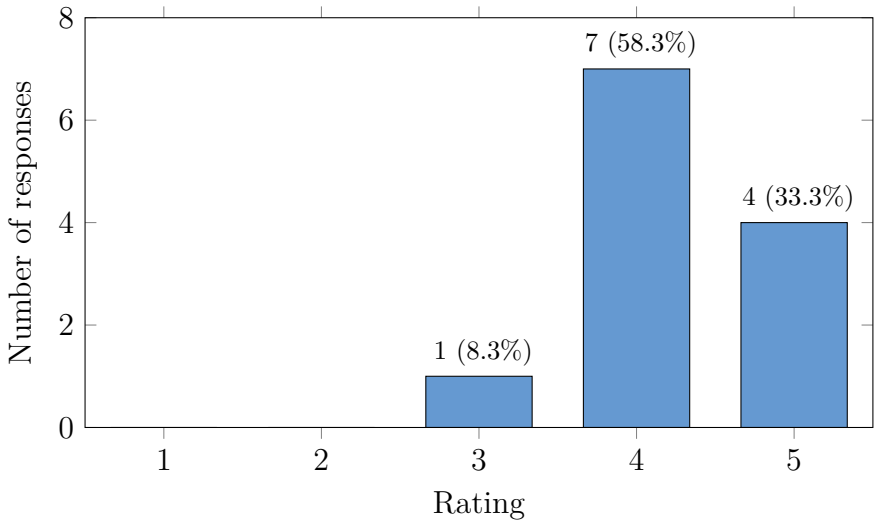


Figure 4.4: A graph depicting the number of responses and the distribution of ratings for the statement "The user interface is visually appealing (colors, fonts, etc.)".

Question: I can easily tell what the different components are supposed to represent

As shown in Figure 4.5, participants could recognize and distinguish the different components, with most participant (10/12) giving the highest rating. However, some participants also commented that this was helped by the accompanying text label for each component, rather that visual design alone. One comment also mentioned that while the components are easily recognizable, the design of them is unappealing.

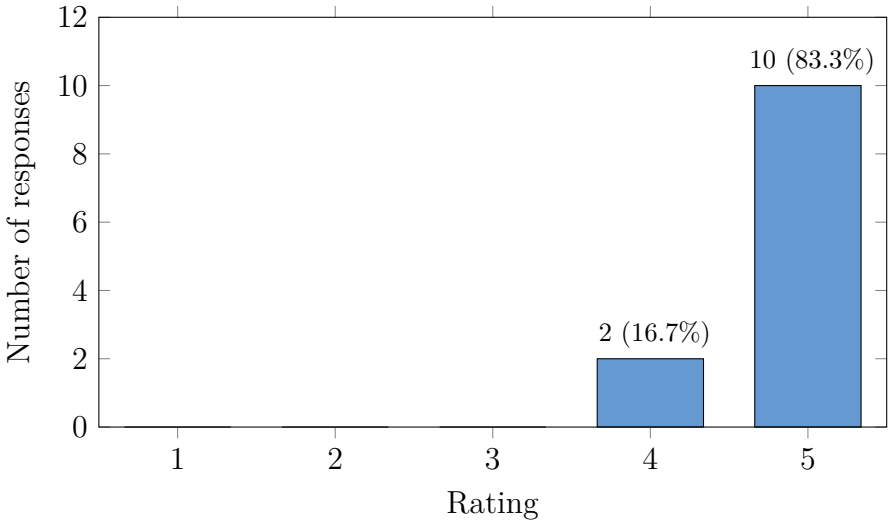


Figure 4.5: A graph depicting the number of responses and the distribution of ratings for the statement "I can easily tell what the different components are supposed to represent".

Question: I think the ability to have different themes would be useful for me

As shown in 4.6, participant ratings for this statement are mixed, but the comments give a clearer picture. Most generally considered the feature nice, but that they would not necessarily use it themselves. One comment wished for more options or the ability create your own themes. Another suggestion was to have a standard light and dark mode.

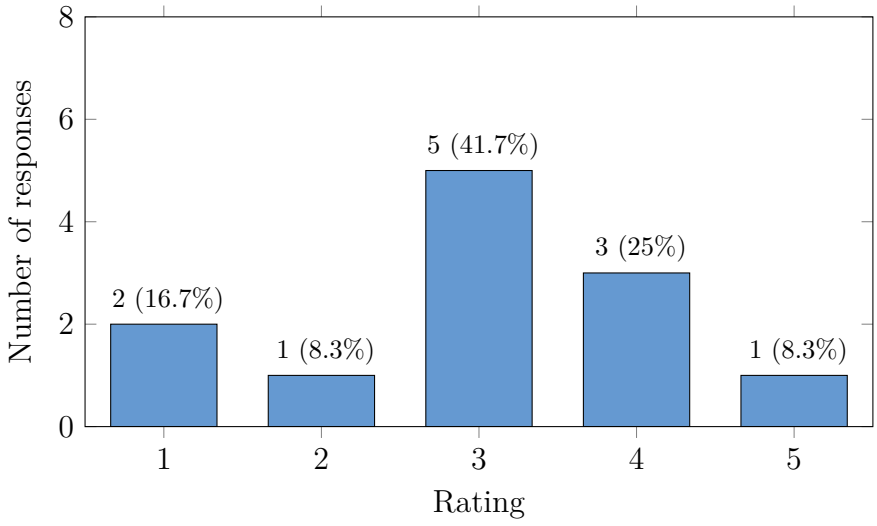


Figure 4.6: A graph depicting the number of responses and the distribution of ratings for the statement "I think the ability to have different themes would be useful for me".

Question: I think the ability to have different themes would be useful for others.

As shown in 4.7, participant ratings for this statement are significantly higher than the previous statement, with 7 of 12 giving the highest possible rating. This means the respondents believe that the option to choose themes could be more helpful for others than for themselves. This is reasonable since most users are not visually impaired or colorblind. The comments to this statement were overall inconclusive, but included one that recognized the potential benefit for visually impaired users, while another noted that the red theme was not colorblind-friendly.

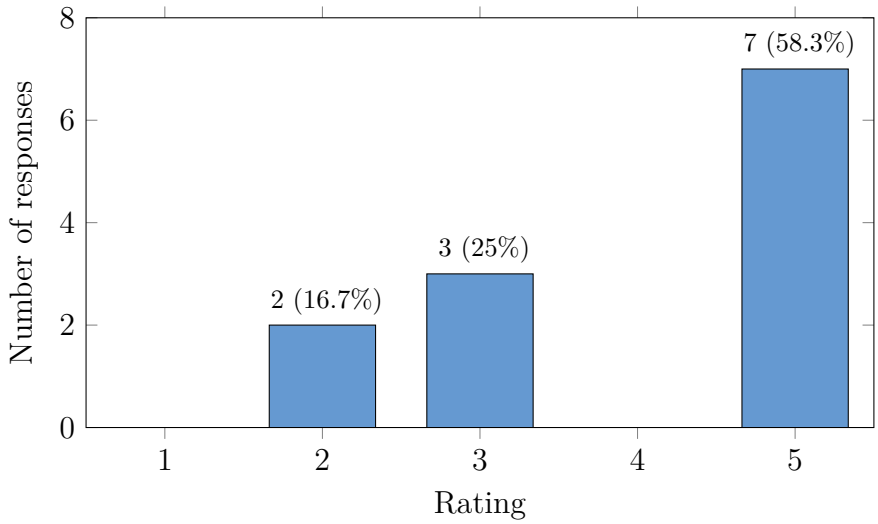


Figure 4.7: A graph depicting the number of responses and the distribution of ratings for the statement "I think the ability to have different themes would be useful for others".

4.3 Educational Benefits

Regardless of visual appearance, the main purpose of the I/O simulator is to provide educational value. To evaluate this, it is compared directly to the old simulator.

Question: I believe that this application would have facilitated my learning in the MOP course

As shown in Figure 4.8, the participants were positive towards the educational value provided by the application, with all ratings distributed above 3. One common opinion was that it would be helpful as long as it was developed further and that the book currently used in the course was updated along with it.

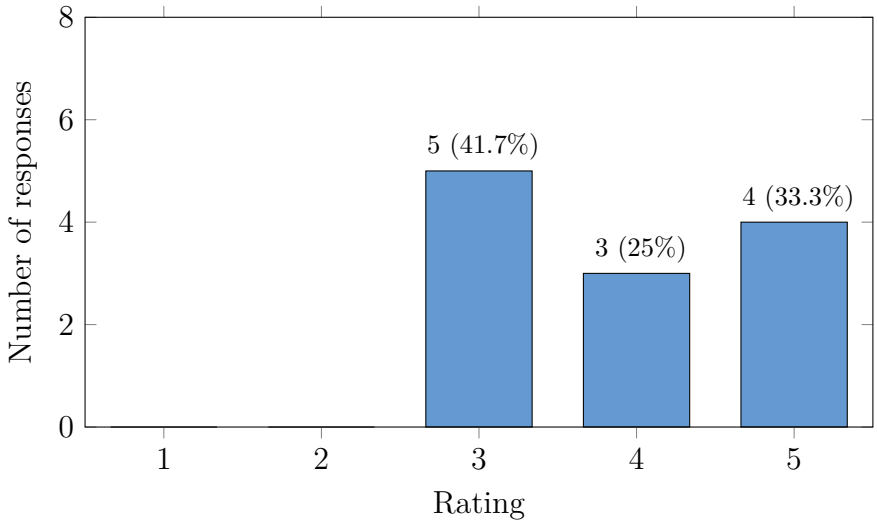


Figure 4.8: A graph depicting the number of responses and the distribution of ratings for the statement "I believe that this application would have facilitated the preparation for the laborations in the MOP course".

Question: I believe that the application would have facilitated the preparation for the laboratory sessions in the course

As shown in 4.9, the average rating shows a slightly more positive attitude towards the usefulness in terms of preparing for the laboratory sessions in the MOP course. Not many participants chose to elaborate on their ratings, but out of the ones that did, three of them were positive, and one was neutral. The comments are similar to the previous question, meaning that the application could be helpful as long as more features are added and the book is updated.

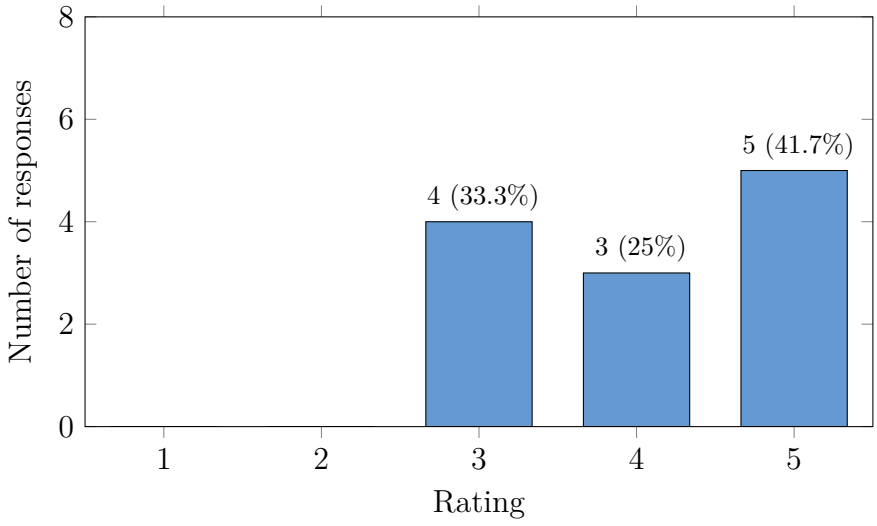


Figure 4.9: A graph depicting the number of responses and the distribution of ratings for the statement "I believe that the application would have facilitated my learning in the MOP course".

Question: I would have preferred to use this application instead of previous tools used in the course

As shown in 4.10, the ratings indicate an overall preference for the developed application over the previous SimServer interface, with 10 of 12 respondents rating it as 4 or 5. Participants expressed that they liked the application, and that they would prefer it over the previous one as long as all peripherals are fully implemented. Most of the comments mention that the new one is a more appealing user interface and that it was easier to use and navigate. There is one outlier who gave the lowest possible rating, but the comment to this rating states that they would use it as a supplement to the current simulator, which is not necessarily a negative review.

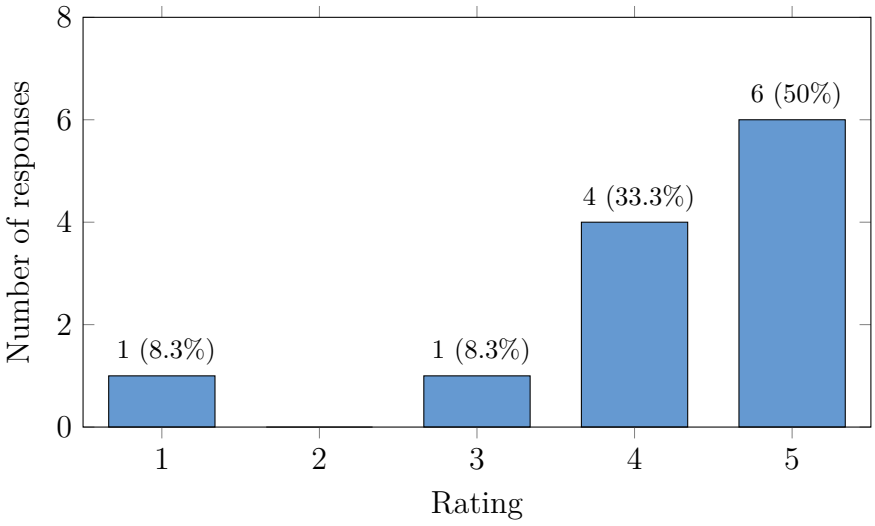


Figure 4.10: A graph depicting the number of responses and the distribution of ratings for the statement "I would have preferred to use this application instead of previous tools used in the course".

5
Discussion

In this section, we will discuss the challenges and decisions made during the development process. This includes the choice of development tools, and the implementation of the I/O Node Editor. We will also address issues related to user feedback and accessibility. Finally, we consider the limitations and how these affected the project. The aim is to reflect on how these factors influenced the final outcome.

5.1 Development Tools

As developers with limited experience in web development, our overall impression of Svelte was that the syntax and structure was simple and easy to learn, and that the extra features like runes and stores were useful for the implementation.

However, a lot of the available information was based on older versions of Svelte, which sometimes caused us confusion and extra debugging. React, which was another option we considered, has a more established ecosystem with significantly more resources and community support available, which could have made it easier to learn despite its more complex syntax.

5.2 I/O Node Editor

The I/O Node Editor was designed as an interactive node editor, as described in Section 3.1.3, because it could more closely emulate the laboratory environment and therefore be more intuitive to students. The main alternative to this would have been more similar to the current SimServer interface, where each peripheral has its own page with an indicator for which port it is connected to. This could have resulted in a more accessible interface, since the node editor does not work with screen readers. Students expressed that the node editor was easier to work with once they understood how, but a menu system similar to the current SimServer interface could have been easier to learn. This could be remedied by adding some kind of guide or tutorial for the application.

Apart from the visual interface for each peripheral I/O unit, each unit also implements a simulation of its functionality. This means that SimServer sends pin states to the application instead of states specific to each peripheral, as described in Section 3.2.4. This works for simple peripherals, but presents problems for more advanced peripherals that depend on timing or that are sensitive to desynchronization caused by WebSocket latency, such as the Keypad. This was solved with a *bridge* in SimServer for the Keypad, but might not work for other problematic units.

An alternative solution would have been to move the I/O simulator to the SimServer backend. This is how the current version of SimServer works. Instead of sending GPIO states to the user interface regardless of which peripheral is connected, the user interface informs the backend of which peripherals are currently connected. This means that the simulation can run without unnecessary delays in the back-end and only peripheral-specific state changes are communicated, which could also increase performance by decreasing the load on the WebSocket connection.

5.3 Interviews

One of the main challenges was the lack of interest from the students to participate in the interviews. The feedback received was very useful, but more participants could have helped us identify the most common issues and user needs. Additional sessions in between the initial user needs interviews and evaluation interviews could have given us continuous feedback on the application, to further assist in guiding the development process.

5.4 Ethics

When developing the application, we wanted to make it as available and inclusive as possible. Previously, many students have found it challenging to practice on their own computers outside of lab sessions. However, with the product being available across multiple platforms, students with different operating systems will have equal opportunities to practice, learn, and succeed in the course. Inclusivity is improved by giving users the option to choose themes, which affects both the color palette and text fonts, which allows both color-blind and dyslexic users to choose a theme that works for them. The simplistic design makes for a less cluttered workspace, which can help users with mild visual impairments.

Systematic integration testing of the I/O simulator together with the MCU simulator and external I/O programs were hard to conduct, which as a result could lead to the application behaving unexpectedly. Worst case scenario could be that users would learn incorrect solutions prior to laboratory sessions.

SimServer is developed by Göteborgs Mikrovaror and is available to download from their website [47]. However, the modified version of SimServer with WebSockets, used in this project, is not publicly available. While this does reduce the reproducibility of this thesis, it does not conflict with the tool's primary purpose as an educational tool in the MOP course, where access would be provided to students if needed. Both SimServer and the new I/O simulator are currently closed-source, which presents some minor ethical concerns regarding transparency and maintainability. Nevertheless, this may be justified by the developers or course administrators, and it does not fundamentally conflict with the software's educational purposes either.

We have spent time and effort into making sure that our data gathering in interviews and forms have been conducted in an ethically responsible way. During both

the early interviews and the evaluation user tests, we made sure that the participants knew that their participation is voluntary. For the early interviews, we asked people who were interested in participating, to fill out a form with their email address and if they preferred an interview face-to-face or via zoom. At the end of the form there was a GDPR clause where we promise to delete their contact information either when the project is done or at latest; 30th of June. We also let the participants know who their personal information will be shared with, and giving them an email address to contact about any questions or if they change their mind and want to opt out of the data collection.

Before conducting the actual interviews themselves, we let the participants decide if they wanted to be in a voice recording or not. After the interviews, we transcribed the interviews and then deleted the recordings afterward. We also did not ask any participants to share their name or any private information while recording. As for the evaluation user test, we did not save any email addresses or names in the form, and every participant was anonymous when answering the questions. Lastly, the I/O simulator itself neither has access to any potentially private information nor does it transmit any sensitive data.

5.5 Limitations

The main limitation was SimServer itself. One reason was the lack of documentation on how it works and how it was implemented. The simulator itself is a very specific program, which means that there is no external documentation. Because the new simulator is still in development, many features have yet to be implemented. This required us to make modifications to SimServer that extended beyond the original scope.

Another limitation was not having access to programs used during laboratory sessions during most of the development process. These would have been useful to test the implementation of each I/O peripheral as a part of the whole system, including SimServer and the communication with our application.

6
Conclusion

In summary, this project has resulted in a web-based I/O simulator that allows users to connect and interact with virtual I/O peripheral units, which is separate from the SimServer MCU simulator. With this, the simulator fulfills its purpose of providing an accessible and educational tool for understanding and experimenting with Machine-Oriented Programming. User tests show that participants generally had a positive experience using the application and its UI, and most expressed a preference for this solution over the current SimServer interface.

For future work, the most important features to implement are the remaining peripherals, which are the ASCII display, Graphic Display, LED, and Buzzer. The system is developed with extensibility in mind, which means that integrating these peripherals should not present any major issues. However, the internal implementations of the ASCII Display, Graphic Display, and Buzzer peripherals could be problematic, since the timing features they rely on have not been tested.

Additionally, the feedback received from the user test was to add a tutorial feature as a general introduction to the I/O Node Editor and individual guides for each peripheral. This would improve the user experience by making it easier to learn. Another aspect of this is the log, which could include more message types from more sources.

A recommendation for how to improve the simulator, and especially for implementing more complex peripherals, is to simulate peripherals in the SimServer backend as opposed to simulating them in the front-end application. This would avoid unnecessary delays and timing considerations associated with the WebSocket communication.

Bibliography

[1] Gothenburg University, *Maskinorienterad programmering*, Accessed: 2025-02-08. [Online]. Available: https://kursplaner.gu.se/pdf/kurs/sv/DIT153.pdf.

[2] Chalmers University of Technology, *Kursplan för maskinorienterad programmering*, Accessed: 2025-02-08. [Online]. Available: https://www.chalmers.se/utbildning/dina-studier/hitta-kurs-och-programplaner/kursplaner/EDA482/.

[3] *Document object model (dom)*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[4] *Typescript: Javascript with syntax for types*, Accessed: 2025-04-22, 2025. [Online]. Available: https://www.typescriptlang.org/.

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice-Hall, 1988, ISBN: 9780131103627.

[6] B. Stroustrup, *An overview of the c++ programming language*, Accessed: 2025-05-01, 2018. [Online]. Available: https://www.stroustrup.com/crc.pdf.

[7] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Pearson, 2015, ISBN: 9780134092669. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/computer-systems-a-programmers-perspective/P200000003634/9780134092669.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann, 2014, ISBN: 9780124077263. [Online]. Available: https://shop.elsevier.com/books/computer-organization-and-design-mips-edition/patterson/978-0-12-407726-3.

[9] David A. Patterson Carlo H. Sequin, *Risc i: Reduced instruction set vlsi computer*, 1981. [Online]. Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S07/handouts/papers/p216-patterson.pdf.

[10] *About risc-v international*. [Online]. Available: https://riscv.org/about/.

[11] *Arm products*. [Online]. Available: https://www.arm.com/products.

[12] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020. [Online]. Available: https://arxiv.org/pdf/2007.03152.

[13] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0 reference manual," Carnegie Mellon University, Parallel Data Laboratory, Tech. Rep. CMU-PDL-08-101, 2008. [Online]. Available: https://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf.

[14] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46. [Online]. Available: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[15] *Digitala läromedel*, Accessed: 2025-04-17, 1989. [Online]. Available: https://www.gbgmv.se/.

[16] Göteborgs Mikrovaror, *Simserver/io-simulator referenshandbok*, Accessed: 2025-04-17, Dec. 2024. [Online]. Available: https://www.gbgmv.se/dl/pdf/Simserver-iosimulator-refman-swe.pdf.

[17] *Learn web development*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development.

[18] *Html: Hypertext markup language*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML.

[19] *Document object model (dom)*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[20] *Css: Cascading style sheets*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS.

[21] Mozilla Developer Network, *Javascript*, Accessed: 2025-05-01, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[22] P. Kirvan and R. Sheldon, *What is a web development framework (wdf)?* Accessed: 2025-05-02, 2025. [Online]. Available: https://www.techtarget.com/searchcontentmanagement/definition/web-development-framework-WDF.

[23] Elementor, *What is a css framework?* Accessed: 2025-05-01. [Online]. Available: https://elementor.com/resources/glossary/what-is-a-css-framework/.

[24] R. Harris, *Virtual dom is pure overhead*, Accessed May 2025, 2018. [Online]. Available: https://svelte.dev/blog/virtual-dom-is-pure-overhead.

[25] N. Kramer, *Svelte compiler: How it works*, Accessed May 2025, 2024. [Online]. Available: https://daily.dev/blog/svelte-compiler-how-it-works.

[26] The Svelte Team, *Reactivity in svelte*, Accessed May 2025, 2025. [Online]. Available: https://svelte.dev/docs/svelte/legacy-reactive-assignments.

[27] The Svelte Team, *Svelte documentation: Stores*, Accessed May 2025, 2025. [Online]. Available: https://svelte.dev/docs/svelte/store.

[28] The Svelte Team, *Introducing runes*, Accessed May 2025, 2023. [Online]. Available: https://svelte.dev/blog/runes.

[29] Meta Platforms, Inc, *Learn react: Describing the ui*, Accessed: 2025-05-02, 2025. [Online]. Available: https://react.dev/learn/describing-the-ui.

[30] Tailwind Labs Inc., Accessed: 2025-05-01. [Online]. Available: https://tailwindcss.com/.

[31] Tailwind Labs Inc., *Optimizing for production*, Accessed: 2025-05-01. [Online]. Available: https://v3.tailwindcss.com/docs/optimizing-for-production.

[32] Skeleton Labs and the Skeleton Community, *Introduction*, Accessed: 2025-05-01. [Online]. Available: https://www.skeleton.dev/docs/get-started/introduction.

[33] Skeleton Labs and the Skeleton Community, *Fundamentals: An introduction to the core concepts of skeleton*, Accessed: 2025-05-01. [Online]. Available: https://www.skeleton.dev/docs/get-started/fundamentals.

[34] The Lucide community, *What is lucide?* Accessed: 2025-05-01. [Online]. Available: https://lucide.dev/guide/.

[35] Skeleton Labs and the Skeleton Community, *Iconography: Learn how to integrate lucide for iconography in skeleton*, Accessed: 2025-05-01. [Online]. Available: https://www.skeleton.dev/docs/integrations/iconography/svelte.

[36] A. Melnikov and I. Fette, *The WebSocket Protocol*, RFC 6455, Dec. 2011. DOI: 10.17487/RFC6455. [Online]. Available: https://www.rfc-editor.org/info/rfc6455.

[37] *Json*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.

[38] S. H. Rogers Y. Preece J., *Interaction Design: Beyond Human-Computer Interaction, 5th edition*. Wiley, 2019.

[39] Interactive Design Foundation, *What is skeuomorphism?* [Online]. Available: https://www.interaction-design.org/literature/topics/skeuomorphism?srsltid=AfmBOop5iGl8FE8q0DOM4erD_fYYSw1Ez3duELyIj4p_3tz5LOjE3FgT.

[40] *Web accessibility initiative*, Accessed: 2025-06-01, 2024. [Online]. Available: https://www.w3.org/WAI/WCAG22/Techniques/general/G174.html.

[41] Accessed: 2025-05-14, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Bezier_curve.

[42] MarianSigler, *Bézier curve.svg*. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/d0/Bezier_curve.svg.

[43] *Getting started with eslint - eslint - pluggable javascript linter*, Accessed: 2025-02-12. [Online]. Available: https://eslint.org/docs/latest/use/getting-started.

[44] *What is prettier? · prettier"*, Accessed: 2025-02-12. [Online]. Available: https://prettier.io/docs/.

[45] *Github - sveltejs/eslint-plugin-svelte: Eslint plugin for svelte using ast*, Accessed: 2025-02-14. [Online]. Available: https://github.com/sveltejs/eslint-plugin-svelte.

[46] *Get started | husky*, Accessed: 2025-02-12. [Online]. Available: https://typicode.github.io/husky/get-started.html.

[47] *Studiematerial*, Accessed: 2025-06-02. [Online]. Available: https://www.gbgmv.se/studies.html#machprog.

A

Appendix 1

Questions used in the interviews: Please note that the interview questions are written in swedish as that was the students prefered language. Questions written in *italics* indicates an example of a follow-up question.

Generella frågor
1. Vilket program/ år går du?
2. Hur mycket har du hunnit använda simulatorn i MOP kursen?

Installation
3. Vilket operativ system använder din dator?
4. Vilken webbläsare använder du främst på din dator?
5. Hur skulle du beskriva processen av att ladda ner och starta upp simulatorn? *Var det lätt/ svårt?*

Simulatorn
6. Kan du beskriva hur du vanligtvis använder simulatorn i den här kursen? *Hemma? På labbar?*
7. Vilka aspekter av simulatorn tycker du är mest användbara? *Lätta att använda/ förstå?*
8. Finns det delar av simulatorn som du upplever som utmanande att arbeta med? *Svårt att använda/ förstå?*
9. Har simulatorn påverkat din inlärning i kursen? *Positivt/negativt?*

UI/ UX
10. Hur skulle du beskriva din upplevelse av att navigera och interagera med simulatorn? *Är det lätt eller svårt?*
11. Kan du komma på tillfällen då du kände dig osäker på hur du skulle gå vidare eller hur något fungerade? *Vad gjorde du då?*
12. Kan du minnas några tillfällen då simulatorn betedde sig oväntat? *Vad hände då?*

Förbättringar
13. Om simulatorn skulle designas om, vilka aspekter skulle du vilja behålla? *Varför?*
14. Finns det några nya funktioner eller förändringar som skulle förbättra din upplevelse?

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS