OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Visualizing and controlling simulated processes using the MTP-concept and OpenBridge

Bachelor Thesis

Submitted in partial fulfillment of the requirements for the Bachelor's Degree Programme in Engineering – *Electrical Engineering and Information Technology with emphasis on automation and medical technology*

AUTHOR(S)

PÅL KRISTIAN OFSTAD
ESKIL G. GAUSTAD
ØRJAN PETTERSEN
FRODE KVALNES

SUPERVISOR

Vahid Hassani

CO-SUPERVISOR(S)

Tor Erik Næbb (WAGO), Håkon Skaug Ingebrigtsen (WAGO)

Bachelor thesis
Oslo Metropolitan University, 2021
Faculty of Technology, Art and Design
Department of Mechanical, Electronic and Chemical Engineering

Acknowledgements

Throughout this project we have received a considerable amount of assistance and support, and would like to thank the people who helped us along the way.

First, we would like to thank Wago Norge AS for giving us the opportunity to work on a solution for a exciting and educational topic that has been very enriching to work with in terms of experience for future jobs.

We would like to acknowledge Tor Erik Næbb from Wago for being our supervisor throughout the project, and for providing us with all the necessary equipment needed to finalize the project.

We would also like to offer our sincere thanks to Håkon Skaug Ingebrigtsen for his unwavering support and patience as our contact person in Wago Norge AS. Not only did he show a great interest in our project, but he also went above and beyond in assisting us at every stage of our project.

Finally, we would like to thank our supervisor from OsloMet, Vahid Hassani, for his valuable counseling and guidance throughout the entire project.

Abstract

Human Machine Interfaces (HMI) has for the longest time been proprietary to a manufacturers own visualization program, often leading to simple and unappealing visuals.
In this thesis, we utilize the OpenBridge Design System to create a website based HMI, removing the limitations set by manufacturers and creating a visually appealing and manufacturer independent HMI.
Simulated processes are created in a PLC by using the upcoming Module Type Packaging (MTP) concept, aiming to establish a manufacturer independent system where the properties of system modules are described regardless of manufacturer and technology.
By using OpenBridge, MTP and OPC-UA, which is a communication protocol between systems, we successfully managed to establish a website based HMI which can communicate bidirectionally with the MTP function-blocks on the PLC.

Contents

Nomenclature

...	In Listings: Complete code not showing
AC	Alternating Current
AI	Analog Input
AO	Analog Output
COG	Course Over Ground
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DC	Direct Current
DCOM	Distributed Component Object Model
DI	Digital Input
DO	Digital Output
FBD	Function Block Diagram
HDG	Heading
HEX	Hexadecimal
HMI	Human Machine Interface
HTML	HyperText Markup Language
I/O	Input / Output
IEC	International Electro technical Commission
IL	Instruction List
JS	JavaScript
LD	Ladder Logic
MTP	Module Type Packaging
n.d	no date

OLE Object Linking and Embedding

OPC-UA Open Platform Communication - Unified Architecture

PC Personal Computer

PEA Process Equipment Assembly

PLC Programmable Logic Controller

PNG Portable Network Graphics

POL Process Orchestration Layer

RGB Red Green Blue

Sass Syntactically Awesome Style Sheets

SCADA Supervisory Control And Data Acquisition

SFC Sequential Function Charts

SOG Speed Over Ground

ST Structured Text

STW Speed Through Water

SVG Scalable Vector Graphics

URL Uniform Resource Locator

VDE English: Association of Electrical Engineering

VDI English: Association of German Engineers

VPN Virtual Private Network

VSclMax alue Scale High Limit

VSclMin Value Scale Low Limit

VSCode Visual studio code

List of Figures

Listings

Chapter 1

Introduction

This thesis and project was made in collaboration with WAGO Norge AS by four students at Oslo Metropolitan University as a final bachelor thesis in regards to the Electrical Engineering and Information Technology study, at the faculty of Technology, Arts and Design.
The objectives of this project has been delivered by WAGO Norge AS with Tor Erik Næbb as the leading engineering supervisor and Håkon Skaug Ingebritsen as the person of contact and support engineer.
The main purpose of this project was to create a functional visual demonstration of a simulated maritime related control system for new potential costumers of WAGO.

1.1 Background

In this section we will explain the background for our project. The background shows why this project was important, and may be a part of the future of automation.

1.1.1 Standardizing Industrial Control Systems

Since the successful launch of Programmable Logic Controllers (PLC) for industrial automation, the issue regarding compatibility towards other systems has been a central topic. In the past, manufacturers had proprietary solutions and programming for their own products. This often resulted in incompatibility between products, time consuming work for maintaining and replacing parts, and a overall poor software architecture. Additionally, large manufactures often gained monopoly in certain cases where the existing system would be too costly for other manufactures to intervene with. In March 1993 a solution to overcome these complications, the first edition of IEC 61131 (International Electro technical Commission), was created with the intention of making standardized requirements for PLC programming, independent of manufacturers (Ramanathan, n.d.).

Compatibility is still a central topic in industrial automation with the the emerging concept of "plug and produce modular automation". Manufacturers today produce systems that are modular, though the automation of these systems is not modular. For example, production equipment like a typical industrial oven is not designed to fit one plant in particular, however it is a modular product that serves multiple purposes for different plants (Wago, n.d.-a).

1.1.2 Visualization

Process control visualizations is often platform dependent and realized in the framework of the manufacturers systems. This tends to make HMI visualizations poorly constructed with little margin for changes and often a unappealing design. As shown in Figure 1.1, how a simple and basic visualization could look in the Codesys based E!COCKPIT program.

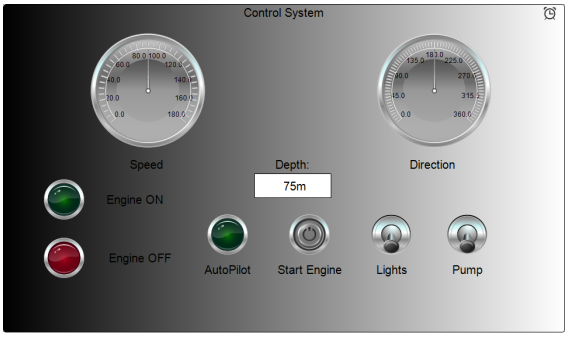

Figure 1.1: PLC visualization

1.2 Project Aims and Objectives

The objective of this project is to design a HMI visualization by following the OpenBridge design concept and industrial standards, while also implementing MTP as a universal interface between the process and visualization. The HMI needs to be able to run on Linux or Docker on a WAGO PLC: PFC200.
To sum up the goals of this project, these are the required specifications:

- Use MTP as a universal interface between the processes and visualization.
- Make a functional demonstration program that Wago can use in sales situations, mainly towards new potential costumers within maritime sector.
- Construct the program as an open system that is compatible with any PLC independent of the manufacturer.
- Design the HMI by using the OpenBridge design concept, and make it in a framework that runs in Docker or Linux on a Wago PFC200 PLC.
- Make a simulated automation process on a Wago PFC200 PLC, with connection to the HMI using OPC-UA.
- Deliver a user manual for the system in English.
- Deliver flow charts and modeling of the processes

Chapter 2

Theory and Practice

In this chapter, all the relevant theory behind the project will be explained like HMI, MTP and OPC-UA, to give a greater overall understanding of our project assignment.

2.1 Human Machine Interface

Human Machine Interface or HMI, is a user interface that connects a person to a machine, system or device. HMI is mostly used in a industrial context, though the term can apply to any screen that interacts with a device. HMI is a important visualization tool that in industrial settings can be used, but not limited to:

- Display Data
- Track time and trends
- Monitor machine inputs and outputs
- Act as a control system

The HMI of a system is often located near the process, but can also be accessed through a VPN. Similar to how someone would control the temperature in their house, an HMI can be used to control the temperature in a industrial water tank. Human Machine Interfaces communicate with PLCs and sensors to display data for users. HMI can either be used as a single function device, only monitoring and tracking, or to perform advanced operations with a specter of different inputs.

The HMI's main objective is to optimize an industrial process by centralizing and digitizing data for users to view. Operators can use the screen to see important data displayed on charts, graphs or dashboards, manage alarms, change controls, and connect with SCADA or other control systems, all through one console (Inductive Automation, 2018).

2.2 Programmable Logic Controller

Programmable Logic Controller is an industrial computer that is often used in industrial automation for manufacturing processes, plants, and other environments. Most PLCs today operates on multiple programming languages, such as:

- Ladder Logic (LD)
- Function Block Diagram (FBD)
- Structured Text (ST)
- Sequential Function Charts (SFC)
- Instruction List (IL)

Control over a PLC system can be achieved by implementing SCADA and HMI systems, which provide the user with an interface that can control the monitored data at the manufacturing floor. The PLC's program works in a cycle, with three main stages: Read input, execute the program and write outputs.
The cycle starts with the PLC detecting the state of the inputs from the connected devices. Then continues to process the data in the CPU, where the program logic is applied to the data from the read input values. Lastly the CPU executes the program logic and writes the output to the connected devices. After each cycle, the PLC does a safety diagnostic, to ensure that everything is operating as instructed (Inductive Automation, 2020).

2.3 Module Type Package

Module Type Package (MTP) is an idea for standardizing module integration in process engineering and manufacturing (Wago, n.d.-a). The goal is to make module integration in new plants as well as existing plants more efficient. The standardized module integration is a framework on how to describe a modular system in the Process Equipment Assembly (PEA), and integration of modules for the Process Orchestration Layer (POL).
PEA is in module engineering specified as a nearly autonomous modular process unit, consisting of one or more functional equipment assemblies, that represents a process or infrastructure within a modular plant (Fachbereich Industrielle Informationstechnik, n.d.). PEA describes a module's physical construction for the processes realised, and required information for integrating it with interfaces in higher level systems, such as the HMI. Process Orchestration Layer is the definition for the elevated control-/visualization system in plant engineering. This includes all the various interfaces and visualization produced in the lower levels of engineering.

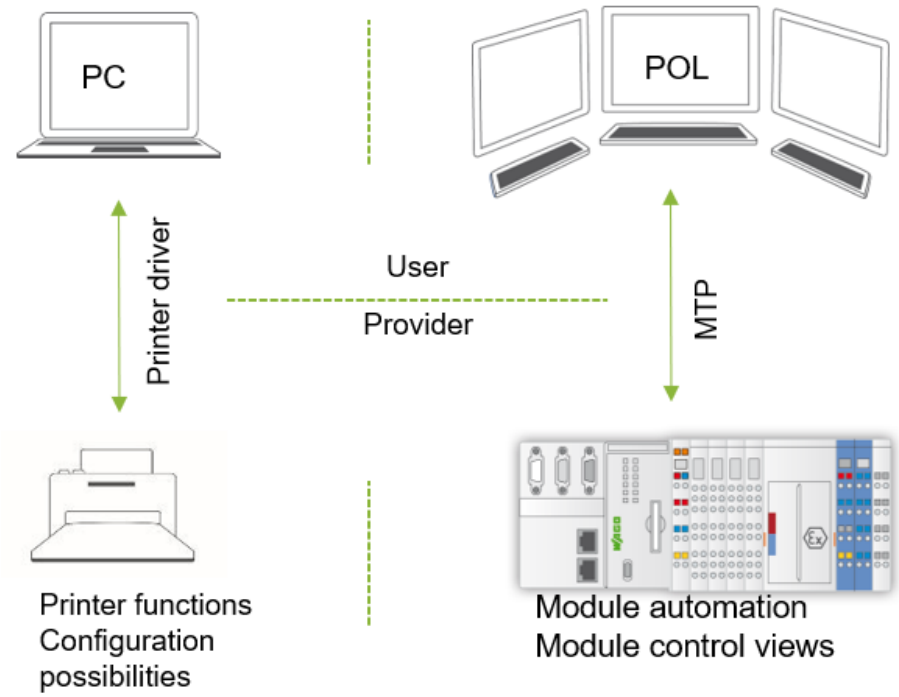

Figure 2.1: MTP concept (Wago, n.d.-a)

The MTP concept is comparable to the setup of a private printer. When installing a new printer, the manufacturer includes the printer driver. This driver includes all the required details regarding the printer, and how to integrate all its services with the computer. In context of a industrial environment, the automation module translate to the printer, the MTP to the driver, and the PC to the control system (POL) (Wago, n.d.-a).

Implementing the MTP framework in the PLC programming would reduce the need for customized programming and installation required for new units or replacing old units in a existing system. This is achieved due to pre-automated modular building units. As MTP framework aims to contain all the necessary information regarding HMI, communication, maintenance information and services, that is needed for integrating modules into modular systems. This would make it easier to later replace components such as a sensor, when subscribing to MTP elements in the POL. This is because the integration would be identical, regardless of the module/equipment manufacturer.

As documented by Fachbereich Industrielle Informationstechnik, n.d., the VDI/VDE/NAMUR 2658 standard is described and produced by the specialist committee "Future Architectures in Automation" of VDI/VDE society for Measurement and Automatic Engineering, in collaboration with NAMUR and the German Electrical and Electronic Manufacturers.

2.4 OPC-UA

OPC Unified Architercture (UA) is a platform independent server/client communication which is based on the OPC Classic specifications. Initially, OPC stood for "OLE for Process Control" which was based on the Microsoft technology OLE and DCOM. OPC UA differs from the classic OPC standard as it is no longer based on OLE or DCOM, which is why OPC now stands for "Open Platform Communication". This makes communication over the network no longer an issue and is what makes OPC UA platform independent. Combining this with secure encryption and authentication technology has made OPC-UA the standard for industrial communication in SCADA systems (Novotek, n.d.).

OPC UA is a client/server communication which means that one or more servers can await requests from one or more clients. When the server receives a request, it sends out that data to the client. The client can also instruct the server to send data when new values are registered, which is called a subscription. With OPC, it is the client that decides which data is sent and received from the server to the underlying system (e.g. PLC) (OPCFoundation, n.d.).

2.5 NodeJS

NodeJS is an Open-Source JavaScript runtime-environment for server and network applications. NodeJS excavates JavaScript code using the Google V8-engine, making it possible to run on servers. A large portion of NodeJS modules are created in JS, though some are made in C# and C++. Modules are often complex functions which is organized in multiple JavaScript files, making it easier to reuse when building applications (OpenJS Foundation, n.d.-a).

Chapter 3

Method and Materials

There were multiple ways to proceed for solving this project, some requested by WAGO and some chosen ourselves. This chapter will thoroughly explain the hardware and software that were used in this project, why we chose to use it, and an explanation on how they can be used.

3.1 Programming in Visual Studio Code

Visual Studio Code (VSCode) is a free to use desktop code editor which is compatible with a wide range of programming languages such as JavaScript, CSS, HTML, Python, Node.js and C (Visual Studio, n.d.-c). VSCode also lets the user install extensions that suits their needs, such as "Live server".
The HTML, CSS, and JS programming files for this project were all created using the Visual Studio Code editor because it is considered to be one of the best code editors on the market, according to an article from creativebloq.com (Smith and Granell, n.d.). This provides a large community for support and discussion to any issues that may occur. VSCode is highly customizable, making it easier to organize the code, and it provides a large quantity of different extensions.

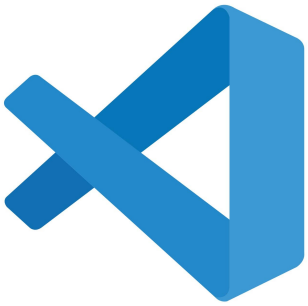

Figure 3.1: Visual Studio Code logo (Visual Studio Code, n.d.-b)

3.1.1 VSCode Extensions

Visual Studio Codes wide range of extensions was one of the main reasons why we chose this code editor, as we could easily add extension continuously when required. Extensions are individual programs that can be installed on the code editor to add specific tools such as languages and compilers. (Visual Studio Code, n.d.-a)

Live Server

For this project we used an extension called "Live server", which is provided by Visual Studio Code (Visual Studio, n.d.-b). This opens the HTML file in the web browser and automatically updates the site whenever a change to the document is made. This makes it very easy to observe how the website reacts to each line of code we add, making it great for efficient coding.

Live Sass Compiler

Another extension that we used was "Live Sass Compiler" (Visual Studio, n.d.-a). This extension lets CSS be written with an extension called Sass, that improves organizing and makes CSS styling more efficient. Live Sass Compiler automatically compiles the written Sass code into CSS, which HTML can understand.

3.2 Website development

The Wago's Touch Panel (TP600) displays the HMI, using HTML5 to structure the web page, Cascading Style Sheets (CSS) to style, and JavaScript (JS) to add dynamic and interactive elements such as buttons and special visualizations to the HMI. Each of the these languages has its own file in the project folder, and they are all imported in the HTML index file.

3.2.1 Structuring with HTML5

HTML5 is the latest version of HyperText Markup Language (HTML). It is the standard markup language for creating websites (W3schools, n.d.). HTML structures the website and tells the browser how to display content (Mozilla, 2021). The content of a HTML document is laid out using "elements" such as <title>, <body>, , <head>, <section>, <article>, <img>, <p>, , <input>, <ul>, <li> and many others. The HTML tells the browser how to display the content based on its setup of "elements".

"Elements" can be assigned attributes that holds more information about the element. There are various attributes in the HTML language, such as "id", "class", "onClick" and "style", that opens a lot of opportunities when styling or positioning elements.

We have used HTML5 to create elements with content such as text, buttons, and icons. To style and position these elements, we have used Cascading Style Sheets (CSS). When creating a HMI that sends and receives data through a server, the website is required to not reload during its use. To accomplish this we had to structure a single-page application that only has one HTML file, and at the same time creates the illusion of multiple web pages.

Figure 3.2: HTML5 logo (w3, n.d.-b)

3.2.2 Cascading Style Sheets

Cascading Style Sheets (CSS) is a programming language used to define the appearance of a website (w3, n.d.-a). While HTML is used to structure the website by placing "containers" such as text or buttons, CSS is used to style those containers, by adding colors or changing fonts and layouts. CSS offers a wide variety of styling, and can even be used to add effects or animations to the website.

3.2.3 OpenBridge

OpenBridge is a new design guideline for maritime workplaces currently under continuous development. This standard has the goal to combine large quantity of different systems from different suppliers in the same user friendly environment, as maritime vehicles are often made with a diverse combination of systems. One standard for all systems also decrease user errors, hours spent on training, and development- and maintenance cost, according to the OpenBridge team (Oslo School of Architecture and Design, n.d.).

OpenBridge mainly consists of two module folders which contains the necessary variables to produce the wanted visualizations. Theses folders are available in GitLab repositories which is found through the OpenBridge website. The first folder is "openbridge css" and is the root of all the components (Bø, n.d.-a). The style sheets and all the variables for the different theme coloring and base components like the navigation menus, are located in this folder. By linking our own components to these variables in the CSS style sheet, we could make them adjust with the theme changes, similar to the already produced components.

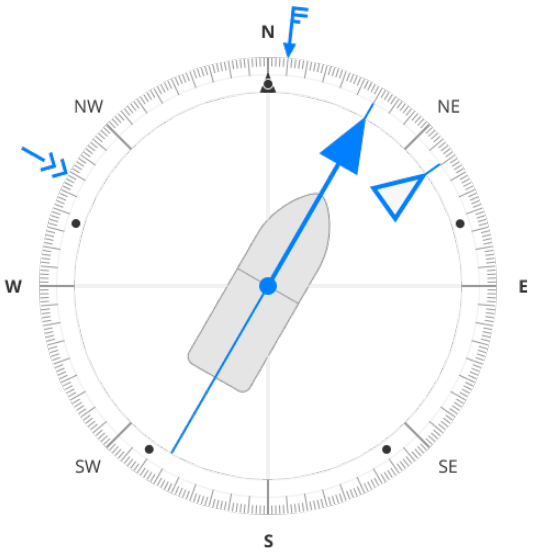

Figure 3.3: Instrument example: Compass (Oslo School of Architecture and Design, n.d.)

The second folder is "OpenBridge web components" (Bø, n.d.-b). This is where the premade OpenBridge instruments compile their visualization and functionalities from. These instruments are compiled through a JavaScript that combines SVG and CSS files to create the visuals, and manages the values displayed according to the given input. To use these visualizations, the OpenBridge team have made an accessible storybook demo page for the already finished components. From the demo page, we could copy the HTML tag for each component and paste it in our index.html file, in similar fashion to this:

```
<ob-hdg-large courseOverGround="211" heading="70" northUp="" ...
    style="height: 612px; width: 612px;" />
```

Listing 3.1: OpenBridge web component example

In this example (Listing 3.1) the name of the component is "ob-hdg-large" and it will display a compass similar to figure 3.3.

3.2.4 Interactivity with JavaScript

JavaScript (JS) is a single-threaded programming language, meaning it can only do one thing at a time. It is used both on client-side and server-side (Hack Reactor, 2018), which allows us to create dynamic and interactive web-pages.
Compared to other programming languages, JavaScript is a simple and easy-to-learn language (Stackify, n.d.), and is the dominant scripting language for making interactive websites.

Figure 3.4: JavaScript logo (The Safety, n.d.)

Javascript was mostly used for developing an interactive front end of the website. Below is an example taken from the (JS) file. These lines of code constructs a function that displays CSS-style on the website when called upon. In this example, the function is called when a button is clicked on the website.

```
function coneDisplay(myDIV) {
  var x = document.getElementById(myDIV);
  var y = document.getElementById(myDIV + "-overlay");
  if (x.style.display === "block") {
    x.style.display = "none";
    y.style.background = "var(--element-global-color)";
  } else {
    x.style.display = "block";
    y.style.background = "var(--element-neutral-color)";
  }
}
```

Listing 3.2: Example javascript code

Plotly.min.js

Plotly is an open source, free to use graphing library available in the languages: Python, R, and JavaScript. It is used to set up intuitive graphs, charts and mappings for statistic analysis, web applications and more (Plotly, n.d.). As this project was aimed at maritime systems, we chose Plotly as it could be used for a real-time depth graph.

3.3 Hardware Assembly

Our project is solved with a variety of hardware and components for our control system. Through this section we will thoroughly explain why we choose the components, and how we assembled and connected the components. A list of all the components is also found in Appendix E.

3.3.1 PFC 200 (750-8212)

PFC 200 Controller is a compact PLC for the modular WAGO I/O System. Besides the processing industry and building automation, typical applications for the PFC 200 include standard machinery and equipment control. The PLC runs on a Linux 3.18 operating system, and supports the programming environments e!COCKPIT (based on CODESYS V3), or WAGO-I / O-PRO V2.3 (based on CODESYS V2.3). The PLC supports programming in the following programming languages: IL, LD, SFC, FBD, ST and CFC (Wago, n.d.-c).

Figure 3.5: PFC200 (Wago, n.d.-c)

This PLC supports a broad variety of different I/O modules, that are used for connecting other components to the PLC. There are tracks for mounting the I/O modules, located on the right side of

the controller. The mounting system enables communication between the modules and the PLC. In addition, the mounting system connects a power circuit for the modules separately from the PLC's power supply circuit. This makes the system more resilient towards power outage. Communication with the PLC is either done with one of its two RJ-45 Fieldbus connections, or with the serial interface RS 232/485 (Wago, n.d.-c).
Ethernet connection is configurable and can also operate as a switch between the network router and the computer. It also gives the opportunity to use the integrated web based management. We chose to use Ethernet connection in this project due to it being the newest, fastest and most supported interface. The PFC 200 is Wago's most sold programmable controller unit, with the 2nd. generation model 750-8212 being their newest edition, surpassing the first generation model 750 8202. As this thesis is in collaboration with WAGO Norge AS, it is by their request that the project is solved with the PFC 200 (Project Aims and Objectives 1.2).

3.3.2 Power Supply (787-2850)

The controller and the modules requires a power supply with 24V DC. We have used the WAGO Power Supply Compact 750-2850. This power supply receives an Alternating Current (AC) signal, ranging from 100-240V, and converts it into three separate 24V DC outputs (Wago, n.d.-c).

Connecting the power supply were done by wiring the hot, neutral and ground input to a 230v supply cable with schuko plug. Then the components could be connected to one of its 24v DC outputs. The analog voltage regulator (3.3.5), the digital switch interface (3.3.4), touch panel (3.3.3) and the PLC (3.3.1) were connected to the outputs of the power supply. Additionally, a parallel loop between the PLC and the I/O module system. When using all the hardware, two power supplies is required.

Figure 3.6: Power Supply (Wago, n.d.-c)

3.3.3 Touch Panel 600 (762-5205/8000-001)

Advanced Line Touch Panel 600 is a 15,6" touchscreen with Linux operating system, and a programming environment for e!COCKPIT (Based on CODESYS V3). Communication with the touch panel is done through one of the two RJ-45 Fieldbus connection. Visualizing on the touch panel is realised through a webserver or e!COCKPIT's WebVisu (Wago, n.d.-c). It is connected and powered by 24V DC supply voltage from the power supply (787-2850).

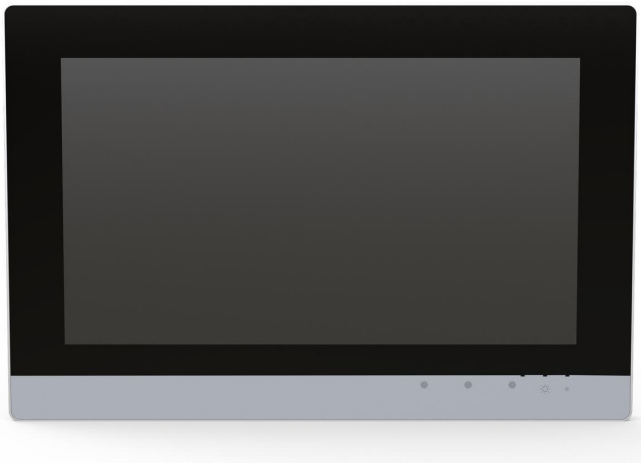

Figure 3.7: TP-600 (Wago, n.d.-c)

3.3.4 Digital I/O Module and Switch Interface

Since the project was merely meant to simulate real system processes used in the maritime industry, equipment that could simulate values were required. After discussing with our supervisor at WAGO, we came to an agreement on certain components for simulating digital and analog values on the PLC.

Digital I/O Module (750-1506)

The I/O module 750-1506 is a digital 8-channel for in/out signals. This module have 8 single wired digital inputs and outputs. When the I/O module is connected, it uses Power Jumper contacts to release the single signal to the ground. This means that the module only needs a single connection for registering a high/low signal. The module shows a green indicator light to know which port is high or low (Wago, n.d.-c).

Figure 3.8: DI/DO module (Wago, n.d.-c)

8-Channel Switch Module (288-853)

The 8-channel switch interface module is a input module with 8 toggle switches, mostly used in test/demo systems. The toggle switches sets the output high or low, with a indicator light showing which port is high or low. The module requires an external power source with 24V DC (Wago, n.d.-c).

Figure 3.9: Switch Interface (Wago, n.d.-c)

Connecting the I/O Module to the Toggle Switch Interface

The I/O module 750-1506 is mounted on the module tracks on the PFC 200. The switch interface (288-853) is connected to the power supply through channel 1 and 2. Channel 3 to 10 is controlled by the toggle switches on the interface. When a switch is high it sets the corresponding output high. The 8 switch outputs are wired to the Push-in CAGE CLAMP inputs on the digital I/O module.

3.3.5 Analog I/O Modules and Regulator

Analog modules is used to receive and transmit analog values. These modules processes the analog values from other components. In this project it is used for simulating analog values for the control system used in the e!COCKPIT software.

Analog Module (750-459) and (750-559)

The I/O modules 750-459 and 750-559 are used for processing analog signals. The module 750-459 have four analog inputs that can process signals between 0-10 Volt. We used it for receiving analog values from the voltage regulator (Wago, n.d.-c).
The I/O module 750-559 have four analog outputs that can send out signals between 0-10 Volt. It is used for receiving data values internally from the PLC and then transmitting the value to the connected device. The two modules shows a red indicator light to know which port is active (Wago, n.d.-c).

Figure 3.10: AI module (Wago, n.d.-c)

Analog Voltage Regulator (286-914)

Voltage regulator (286-914) is a regulator that can adjust the voltage over the component by adjusting its internal resistance. The output can be regulated between 0-10V. An analog regulator module is mostly used for test/demo systems. The module requires an external 24v DC power circuit (Wago, n.d.-c).

Figure 3.11: Voltage regulator (Wago, n.d.-c)

Connecting the I/O Module to the Regulator

The I/O modules 750-459 and 750-559 were mounted on the module tracks on the PFC 200. The analog voltage regulator (286-914) was directly mounted on a connection clamp, which was connected to the power supply circuit. Further, the two outputs on the regulator connects to the Push-in CAGE CLAMP inputs on the I/O module 750-459.

3.3.6 End Module (750-600)

The I/O module 750-600 is an end module, and a part of the PFC 200. It ends the I/O circuit and protects data flow (Wago, n.d.-c). The I/O modules: 750-1506, 750-559 and 750-459 were mounted on the right side of the PFC 200, and isolated with the end module (750-600).

Figure 3.12: End Module (Wago, n.d.-c)

3.3.7 Misc

Other equipment or hardware used in this project were:

- 2.5mm Flathead operating tool for attaching the cables to the Push-in CLAMP CAGE on the I/O modules.
- Cable-cutters for cutting the lab cables.
- Connection cable for connecting PLC I/O modules to the components and power supply.
- ETHERNET cable Cat 6 RJ45 for communication between the PFC 200, touch panel 600 and computer.

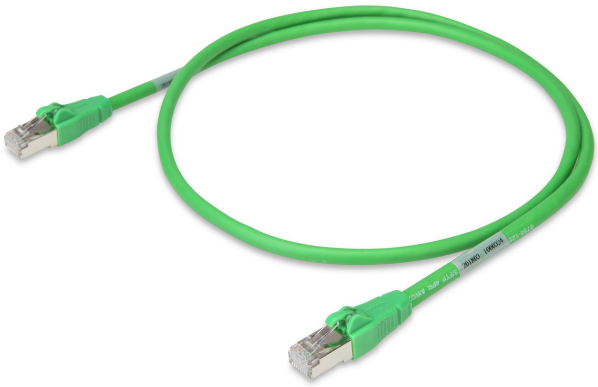

Figure 3.13: TP-600 (Wago, n.d.-c)

3.4 PLC

A requirement for this thesis was that the PLC supports OPC-UA communications, in order to establish connection between the PLC and the HMI. Our project required the PLC program to read, write and send values to the web interface. Since the style of the interface was aimed towards the maritime industry, the program functions take the form of different maritime systems. Such systems are: navigational steering system, engine control, tank control and light control. A key aspect of this thesis regarding the communication between the OPC UA server and client, was describing the PLC functions within the MTP framework standard VDI/VDE/NAMUR 2658 Blatt 3 (Project Aims and Objectives 1.2).

3.4.1 e!COCKPIT

e!COCKPIT is WAGO's own CODESYS V3 based automation software. It is an all-in-one integrated development environment that supports hardware configuration, simulation, programming, visualization and OPC-UA communication (Wago, n.d.-d). Wago has pre-built MTP functions within the framework standard VDI/VDE/NAMUR 2658, in the library extension "e!COCKPIT MTP Package Version 1.0.1. From the thesis collaboration with WAGO Norge AS, e!COCKPIT was chosen as the default software for PLC programming when solving this aspect of the project.

Figure 3.14: e!COCKPIT Logo (Wago, n.d.-b)

Using MTP elements

Manually controlled hardware has to be integrated with indicator elements from the MTP library. Such MTP indicator elements are AnaView and AnaMon. AnaView is a indicator element used to view the analog value of a process, with parameters for minimum and maximum values. AnaMon is an extension to AnaView, that can set multiple alarms and warnings (Fachbereich Industrielle Informationstechnik, n.d.). We found the alarm limits useful for the control processes, due to elements on the web page that required both monitoring and alarms.

Listing 3.3 shows an example of how a real number input from a temperature sensor could be integrated with an AnaMon element. Limit checks between the initial Value Scale Low Limit (VSclMin) and Value Scale High Limit (VSclMax) are made in the variable declaration. These limit checks are pre-defined offsets for the alarms and warnings. Value for Alarm Low (VAL), is one alarm that is either enabled or disabled depending on the process. When enabled it gives a high output "VALAct" if the input is lower than its limit.

```
PROGRAM ANAMON_Example
VAR
rEngineTemp: REAL; //Engine temperature sensor
EngineTemp_Mon: AnaMon:=(
 VAHLim          := 190, //Limit Value for Alarm High
 VWHLim          := 170,//Limit Value for Warning High
 VTHLim          := 150, //Limit Value for Tolerance High
 VTLLim          := 10,  //Limit Value for Tolerance Low
 VWLLim          := 0, //Limit Value for Warning Low
 VALLim          := -10);  //Limit Value for Alarm Low
END_VAR



EngineTemp_Mon (
 TagName         := 'monitor engine effect',
 TagDescription  := ,
 V               := rEngineTemp,  //input from temp sensor
 VSclMin         := -10, //Sets Lower Limit In Celsius
 VSclMax         := 200, //Sets Higher Limit In Celsius
 VUnit           := WagoSolMTP.eUnits_IEC61158.degree_celsius,
 VAHEn           := TRUE,//Enables Alarm High Limit; 1:enabled; 0:disabled
 VWHEn           := TRUE,//Enables Warning High Limit; 1:enabled; 0:disabled
 VTHEn           := TRUE,//Enables Tolerance High Limit; 1:enabled; 0:disabled
 VTLEn           := TRUE,//Enables Tolerance Low Limit; 1:enabled; 0:disabled
 VWLEn           := TRUE,//Enables Warning Low Limit; 1:enabled; 0:disabled
 VALEn           := TRUE,//Enables Alarm Low Limit; 1:enabled; 0:disabled
```

Listing 3.3: Using hardware with MTP elements

3.5 Communication

When deciding on how we would tackle the communication between all devices and programs, we divided the communication into different parts. We knew the PLC utilized OPC-UA and that it would need to connect to a client, as this was one of the requirements set by WAGO (Project Aims and Objectives 1.2). We thought that the HTML would also need to get hosted from a local script which could communicate bidirectional. After a little thought process and discussion, we created a simple flow chart:

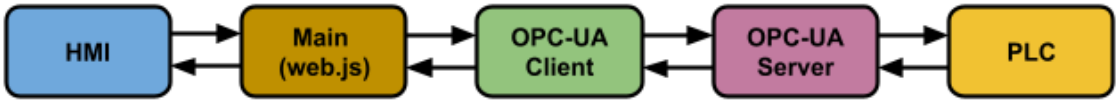

Figure 3.15: Thought process

The idea, as shown in Figure 3.15, based itself on dividing the communication into small groups, making it easier to focus on the different parts.

- Firstly, the HTML could be started from a script (Main) and communicate using socket communication.
- The Main-script could also start the OPCUA client and create a communication channel between the two.
- The OPCUA-client could then connect to the OPCUA server hosted from the PLC, sending and receiving variables.
- The OPCUA-server can then send/receive these variables from the PLC, thus completing the connection.

3.5.1 NodeJS

To establish a communication channel between all links (Figure 3.15) we needed a main script which would start and communicate with both the HMI and the OPC-UA client. The script needed to be able to act as a server for the website, as well as a parent process for the OPC-UA client. When deciding on what we should use to program our scripts for the communication channel, we did some research and after a small conversation with our supervisor we settled on using NodeJS (see NodeJS 2.5). NodeJS would work great for our communication channel as it was based on JavaScript, making it compatible with our website. NodeJS has a wide number of different modules (OpenJS Foundation, n.d.-a), and for our main server we decided to use the following node-modules based on their easy to use functions and good documentation:

- **Express**: A NodeJS framework for web applications that allows simple and clean launches of webservers and APIs (ExpressJS, n.d.).
- **Http**: A built in module which allows the transfer of data over http, create http servers with assigned ports and respond to client requests (OpenJS Foundation, n.d.-d).
- **Socket.io**: A module which enables real-time communication between devices, browsers and platforms (Socket.IO, n.d.).

- **Child_process**: The child process module enables one process (parent) to create another process (child) and establish a communication channel between the two (OpenJS Foundation, n.d.-b).

3.5.2 Socket.IO and Child_Process

We chose Socket.IO and Child_Process to be our main ways of communicating between scripts, other than OPC-UA. To get a complete understanding of the inner workings of the communication, here is a brief explanation.

Socket.IO

By using the "io"-function, certain websocket methods for sending and receiving data can be utilized. The "io.emit" and "socket.emit"-functions will send a message in the form of a string to all connected users. These functions also requires a identifier string to ensure the message only reaches the correct destination. The "socket.on"-function is on the receiving end of the emitter function and only receives messages with a matching identifiers as itself, as shown in Listing 3.4.

```
//Server sending a message with identifier "msg1"
io.emit('msg1', "This is a message");
--------------------------------------------------------------------------------
//Client receiving the message
socket.on('msg1', msg => {
    //Receives the message and puts in in the variable "msg"
});
//Client not receiving the message
socket.on('msg2', msg => {
    //Does not receive the message as it doesn't have matching identifiers
});
```

Listing 3.4: Example use of socket.io

Child_Process

Child_Process is a node-module for executing external scripts from a running script. The running script will function as a "parent-process" and the executed script will be a "child-process", hence the name. The Child_Process module has 4 main functions for executing external scripts: execFile(), exec(), spawn() and fork() (Buna, 2017). In this project we chose to use the fork()-function as it is a function for spawning a node and establishing a communication channel between the parent and child.

A child-process can be created by calling the fork()-function and inputting the relative address of the desired script in between the parentheses. Using the **send** and **on** functions together with the global **process** object, the scripts can exchange messages with each other as shown in the example below (Listing 3.5).

```
//Parent-process sending a message to child-process
const child = fork('./child.js');
child.send("Hello Child");
-------------------------------------------------------------------------------
//Child receiving the message
process.on('message',(msg) => {
    console.log('Parent message: ', msg);
});
```

Listing 3.5: Example use of Child_Process

3.5.3 OPC UA

OPC-UA is a server/client communication protocol which is widely used as an industry standard for its security and reliability (see OPC-UA 2.4). OPC-UA is the standard communication for most WAGO PLCs, including PFC200 (750-8212) (see PFC200 3.3.1) which was the PLC used in this project. One of the requirements from WAGO was that we used OPC-UA to integrate automation processes with a visualization (see Project Aims and Objectives 1.2). This meant that we needed to create a OPC-UA client for our project, as the PFC200 would act as a server. The client needed to be able to communicate with our HMI, which was based on a HTML website.

Choosing a client

There were several different ways we could go when choosing a client framework. We took a look into Python, Node-RED and Node-opcua, but settled on the latter (see Discussion 6.7).
Node-OPCUA (Rossignon, n.d.-b), is a typescript written OPC-UA stack for NodeJS. As Node-OPCUA was based on the NodeJS-framework, which again was based on JavaScript, this meant it would be compatible with HTML based websites that utilized external CSS and JavaScript. Node-OPCUA works like a function add-on for NodeJS, meaning we could write our code like normal JavaScript, but use certain functions from the OPC-UA node-module. Coding directly in NodeJS would lead to a lot more freedom, but also difficulty. There were not a lot of information to be found on node-opcua, as it was a fairly unknown node-module with few to no examples of implementation. There was however one example of a simple client on the official node-opcua GitHub-page (Rossignon, n.d.-b), which would be sufficient enough to continue building on.

3.5.4 Ignition

To test the OPC-UA connection, we used a software called Ignition. Ignition is a development environment used to create industrial applications (Inductive Automation, n.d.), which additionally could be used to as a OPC-UA client to visually browse through variables on PLCs and similar devices. We found a tutorial that precisely explained how to set up Ignition as a client for a WAGO PLC (Braun, 2016). Ignition had a free trial period which we used to the fullest when creating our own OPC-UA client.

Figure 3.16: Ignition logo (Inductive Automation, n.d.)

3.6 Finalizing the project

When the software development and hardware assembly was completed, the finished program needed to be uploaded to the Wago Touch Panel and the PFC 200 so that it could run independently of internet connection and PC connection.
To upload the website files to the TP-600, we chose to use a packaging program called Docker (Project Aims and Objectives 1.2) to pack the files in a certain form, and a serial console named PuTTY to access the touch panel and download the files.

3.6.1 Docker

Docker is a software that makes it easy to build, run, manage and distribute different applications. It virtualizes a linux operating system on a computer and creates containers that allows developers to pack an application into it (Docker, n.d.), and then open it on another system with all its parts unflawed. Docker aims to make it possible to run any application on any operating system, given it supports docker.

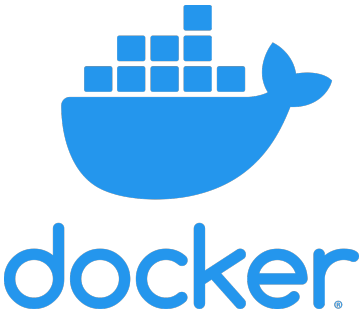

Figure 3.17: Docker Logo (Docker, n.d.)

When using Docker, all the necessary files needed for a website to work are packed and stored in a virtual container running on Linux. This makes the performance of the application significantly better and reduces its size. The container is then packed as an image and uploaded to the Docker cloud, ready to be downloaded to devices such as the TP-600.

3.6.2 PuTTY

PuTTY is a terminal emulator application that supports multiple connection types such as Raw, Telnet, Rlogin and SSH (PuTTY, n.d.). There are many other programs that works the same way as PuTTY, but since we had already learned to use PuTTY at a previous subject at OsloMet, it suited us well to use it for our project.

We used PuTTY to access the computer console of the Wago Touch Panel, keeping track of the CPU performance, and to download and install programs locally on the TP-600's computer. To be able to download and unpack docker images on the touch panel, we had to download and install the Docker program (docker.ipk) from GitHub, provided by Wago (WAGO Norge AS, n.d.).

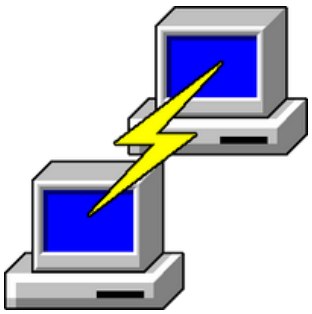

Figure 3.18: PuTTY logo (Behance, n.d.)

Chapter 4

Development

This chapter addresses the different parts of the development of the project. Since the web development alone consists of approximately 3500 lines of codes, the general type of code will be described once, even though it it used multiple times on different elements. The functions with more unique features, will be explained separately. The full program with all files is available in the GitHub repository in Appendix A.
As none of the participants on this project had any previous experience with website development, the first step was to learn the programming languages that we would need to make a website. After a few google searches, it was easy to understand that a website was built up by a markup language (see Structuring with HTML5 3.2.1), a styling language (see Cascading Style Sheet 3.2.2) and a scripting language (see Interactivity with JavaScript 3.2.4), that together displays an interactive website in the web browser. To learn the basics of this type of programming, we mainly followed along with the video tutorials created by the two YouTube channels; "thenewboston" (Roberts, n.d.) and "The Net Ninja" (Pelling, n.d.) as well as reading websites such as "W3Schools.com" (W3schools, n.d.) and forums for programming related questioning.

After we had a good enough grasp around the basics of website programming, we begun to tinker with implementing the OpenBridge visuals, in addition to start experimenting with a connection between a simple website and the PLC.

4.1 Web development

To create a fully functioning and interactive website that looks appealing to the user, we have used two languages that communicates with a HTML5 file; CSS and JavaScript. For the code to run correctly, the different languages, HTML5, CSS and JavaScript needed to be loaded in a certain order, or else the website could end up with multiple errors.
Figure 4.1 shows how the different files are loaded together, and in the correct order. The first file to load is the HTML5, then CSS, and lastly JavaScript. As each file is loaded, they are converted into a program that the browser is able to run and display as a readable website for the user.

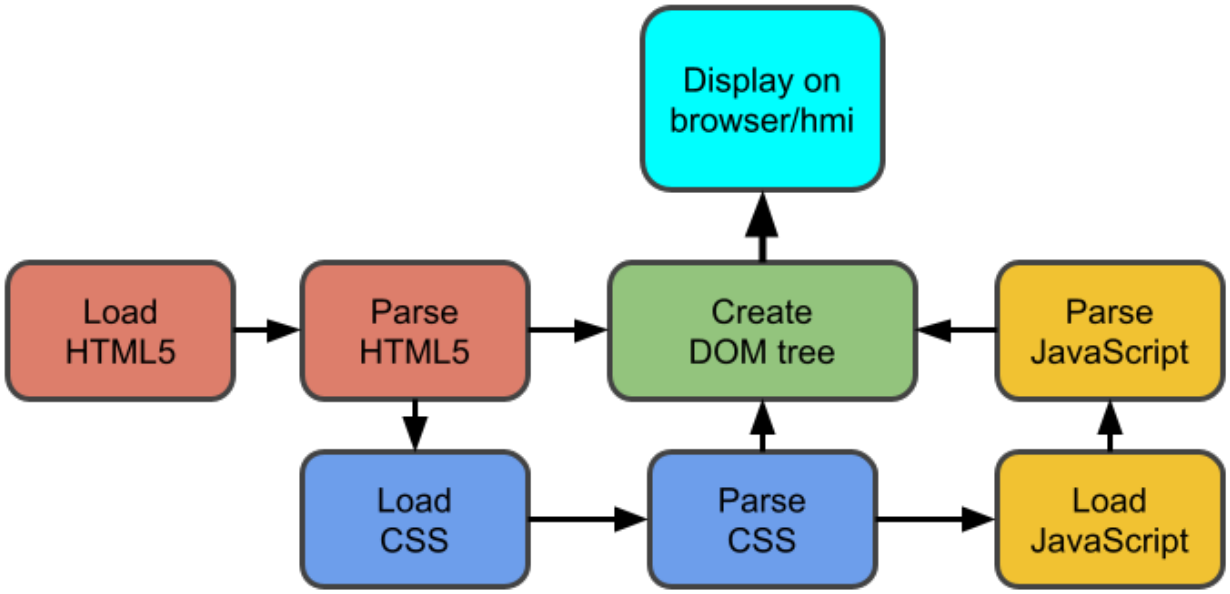

Figure 4.1: Illustration of loading order inspired by diagram from MDN Web Docs, n.d.

4.1.1 HTML5 Structuring

The HTML document is constructed by two main containers; the <head> and <body> elements, where each consists of their own elements that together structures the website. The documents starts with a document-type declaration <!DOCTYPE html> that informs the browser what type of text it is reading. The <head> and <body> elements are located inside a HTML-container with an attribute named "theme", that is given a value "dusk" as seen in Listing 4.1. The HTML-element acts as the root element of the HTML document.

```
   1   <!DOCTYPE html>
   2   <html lang="en" theme="dusk">
   3
   4   <!-- Head-element: -->
   5 > <head>···
  16   </head>
  17
  18   <!-- Body-element: -->
  19 > <body onload="initialFunction()" class="ob-global-surface">···
1026   </body>
1027
1028   </html> <!-- End of code -->
```

Listing 4.1: Shortened extract from html document

The "theme" attribute attached to the HTML-element decides which theme-color the website will display. This is further explained in OpenBridge 3.2.3.

The <head> element

The <head> element of the HTML document serves as a container for metadata. Metadata is not displayed, but defines data such as styles, scripts and the title of the document. The <head> element contains two <link> elements, one <script> element, and one <title> element. The <link> elements links to two different CSS documents; "openbridge.css" and "styles.css". These are further explained in CSS Styling 4.1.2. The <script> element defines a JavaScript file from OpenBridge; main.mjs, that is explained in Javascript 4.1.3, and the <title> element gives the document the name "Bachelor". Listing 4.2 is a extract from the HTML document that shows the <head> container top to bottom.

```
<head>

  <title>Bachelor</title>
   <!-- Open-Bridge Components -->
  <link rel="stylesheet" type="text/css" ...
     href="modules\openbridge-css\dist\css\openbridge.css">

  <!-- Custom Styles -->
  <link rel="stylesheet" type="text/css" href="styles/css/styles.css">

  <!-- Web-Components -->
  <script src="modules/openbridge-web-components/dist/main.mjs"></script>

</head>
```

Listing 4.2: Head-element from html document

The <body> element

The <body> element serves as a container for all the elements that structures the website. Here are all the containers that hold everything we see on the website. These containers are mainly elements named , but other elements such as , <a>, <input> and <form> are also used but not quite so often. Listing 4.3 is an extraction from the HTML document that shows the layout of the account menu in Figure 4.2. As seen from the HTML extraction, the account menu is built out of layers of elements and one <input> element.

```
<!-- Account Menu - Start -->
<div id="accountTab" class="ob-nav-menu theme-color">
    <div class="account-text">Account</div>

    <div id="login-info" class="input">
      <br> <br>
      <input type="text" name="User" placeholder="Username">
      <br> <br>
      <input type="text" name="pass" placeholder="Password">
    </div>

    <div class="signin-btn-frame">
      <div class="signin-btn-text">Sign in</div>
    </div>
</div>
```

Listing 4.3: Account menu layout from html document

The classes that are attached to the elements in the <body> container are given certain colors and positioning using the CSS file that is imported in the <head> container.

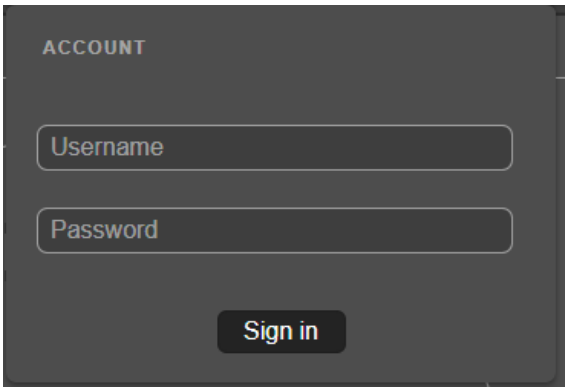

Figure 4.2: Example: account login

4.1.2 CSS Styling

As mentioned earlier, HTML5 does not provide any styling to the website, it only structures it, meaning it only displays the elements you want to display. Without styling the elements in the HTML5 document, the website would look very simple and would not be very appealing to the user. Adding CSS to the website opens many possibilities when it comes to styling. Figure 4.3 shows the vast role CSS plays when designing a website, where the top of the figure is our website without style, and the bottom is with style.

Figure 4.3: Website with and without CSS

The CSS file "styles.css" is included in the head of the HTML document. This loads all the content of the CSS file into the HTML document. To style an element in the HTML document, we have to add a "class" or "id" to it, as shown in Figure 4.4 where two "div" elements are given an "id" and a "class". These classes needs to be given style properties in the CSS file, as shown in the example listed below (Listing 4.4).

```
<!-- Account Menu - Start -->
<div id="accountTab" class="ob-nav-menu theme-color">
    <div class="account-text">Account</div>

    <div id="login-info" class="input">
      <br> <br>
      <input type="text" name="User" placeholder="Username">
      <br> <br>
      <input type="text" name="pass" placeholder="Password">
    </div>

    <div class="signin-btn-frame">
      <div class="signin-btn-text">Sign in</div>
    </div>
</div>
<!-- Account Menu - Slutt -->
```

Figure 4.4: AccountTab example

```
#accountTab {
    width: 359px;
    height: 248px;
    left: auto;
    right: 112px;
    top: 57px;
    border-radius: 7px;
}

.account-text {
    position: absolute;
    width: 311px;
    height: 24px;
    left: 24px;
    top: 16px;
}
```

Listing 4.4: Styling classes and ids

The result of these styles in combination with OpenBridge coloring can be seen in Figure 4.2. This is how the CSS works for every elements in the HTML document.

4.1.3 JavaScript (main.js)

As previously mentioned we use JavaScript to make the website interactive and behave the way we want it to in relation to our input. This allows us to make both simple and complex functions that manipulate the elements on the page to e.g., show, hide, change color or move them.

Figure 4.5: Home page

Navigation menu

The first interactive part that were made was opening the main navigation menu (Figure 4.6). This is where the user would switch between the different pages that they would want to view at any given time. Therefore this is where the different view perspectives of the system were organized.

Figure 4.6: Development: Navigation Menu

```
//Opening and closing drop-down menu
function menu_btn_pressed() {
    this.menu_btn_toggle = !this.menu_btn_toggle;
    console.log(menu_btn_toggle);
    if (menu_btn_toggle) {
    var menubtn = document.getElementById("menubtn");
    menubtn.classList.add("ob-active");

    var sideNavMenu = document.getElementById("side-nav-menu");
    sideNavMenu.classList.add("ob-active");
  }
    else if (!menu_btn_toggle) {
    var element = document.getElementById("menubtn");
    element.classList.remove("ob-active");

    var sideNavMenu = document.getElementById("side-nav-menu");
    sideNavMenu.classList.remove("ob-active");
  }
}
```

Listing 4.5: Navigation Menu opening function

By pressing the menu button in the top left corner, the "menu_btn_ pressed" function in Listing 4.5 would initiate. When pressed, the "menu_btn_toggle" variable changes state from what it previously was. Since it is initially declared as "*FALSE*", it would change to "*TRUE*" and initiate the IF-statement that adds or removes the "ob-active" CSS class to both the button and the "side nav menu". This class is the integrated activation class for OpenBridge elements, therefore since the menu button and the navigation menu are both in the OpenBridge library, we could use it to activate these elements. For the menu button, this class would highlight it and change the colors, as shown in Figure 4.6. For the navigation menu itself, its class parameter "display" would change value from "none" to "block", which would essentially move the menu from a hidden- to a displayed state. This principle is the same on all the elements on the website that are shown and hidden at different times.
If the button was to be pressed again while the menu was open, the same function would initiate, but since "menu btn toggle" now would be "*TRUE*", it would change back to "*FALSE*" and follow the else-if-statement instead. This would do the exact opposite and remove the "ob-active" class from both elements.

Since the "ob-active" class is placed inside the premade OpenBridge elements in the SASS file, we could not use it on the elements that we created ourselves. Therefore we had to make our own "active-class" as well, which was easily done by giving all the elements that we wanted to be able to hide and show a style-property of: "display:none;" and then add a class with a "display:block;" property through a function when we want it to show.

Listing 4.6 shows the functions that made it possible to move through the different pages on the navigation menu. In total there where three functions involved, including the previously mentioned "menu_btn_pressed". The reason for this was that when we selected a page that we wanted to visit, we also wanted the navigation menu to close itself. The same applies to the current viewing page, which was closed with the "closeTab" function. This function looked through all the page elements and found the ones with a given class name, in this instance, we wanted to remove the "displayTab" class. To open the page that was clicked, the "openTabs" function continues and adds the "displayTab" class to it via it's unique element id.

```
//Takes id-tag of the tab that is clicked and adds displayTab-class
function openTabs(tabClicked) {
  menu_btn_pressed();
  closeTabs("displayTab");
  //Adds tab-name to the top navigation bar

  document.getElementById("sectionName").innerHTML = tabClicked;

  console.log(tabClicked);
  var id = tabClicked + "Tab";
  console.log(id);
  document.getElementById(id).classList.add("displayTab");
}
//Goes through all elements and removes the class
function closeTabs(className) {
  var displayed = document.getElementsByClassName(className);
  for (i = 0; i < displayed.length; i++) {
    displayed[i].classList.remove(className);
  }
}
```

Listing 4.6: Switch pages

Depth chart

In regards to the JavaScript, one of the most challenging parts was the real-time depth chart display (Figure 4.7) on the home page. Unlike the other elements, this would need continuous updating both on the traceing itself and the x-axis. The visual goal for this graph was to have it slide left and out of the frame, while new depth readings would plot on the right. Essentially we would have a visual representation of the seabed as the ship would move over it. After some research we decided to use Plotly for this task (See Discussion 6.4).

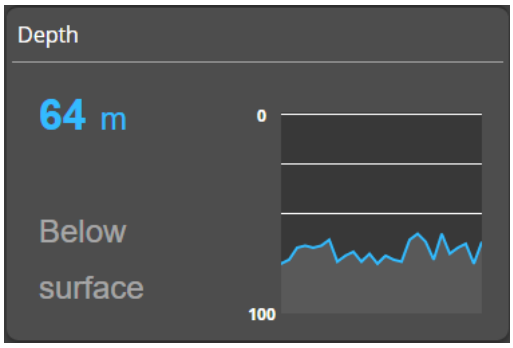

Figure 4.7: Depth Graph

Setting up a static graph with Plotly was quite simple when following the instructions on the their website (Plotly, n.d.). For the graph to be displayed properly, we had to use a variable for the

tracing and a variable for the layout, and then place these variables in the "Plotly.plot" function. This would then be referenced to an HTML element by id. In our case the id is "chart".

```
//Depth graph
function getData() {
  var value = Math.floor(Math.random() * (75-60+1))+60;
  document.getElementById("actual-depth").innerHTML = value;
  return value - 100;
}

var trace1 = {
  y: [getData()],
  fill: "tozeroy",
  fillcolor: getColor("--instrument-track-color"),
  line: { color: getColor("--instrument-dynamic-color") },
};
var layout = {
  width: 150,
  height: 150,
  paper_bgcolor: "rgba(0,0,0,0)",
  plot_bgcolor: getColor("--instrument-frame-color"),
  margin: {
    l: 0,
    r: 0,
    b: 0,
    t: 0,
  },
  xaxis: {
    showgrid: false,
    showticklabels: false,
  },
  yaxis: {
    color: getColor("--element-active-color"),
    range: [-0, -101],
    showticklabels: false,
    gridcolor: getColor("--element-active-color"),
    dtick: 25,
  },
};
var data = [trace1];

Plotly.plot("chart", data, layout, { displayModeBar: false });
```

Listing 4.7: Depth Graph

To simulate a realistic variety in the distance between the ship and seabed, we created a function that would output a random value between 60 and 75 meters. The specific reason for this range was to get some continuity of the depth and not have a sudden change from 0 to 100 meters. This value is called upon in the tracing variable, represented in "trace1", along with with styling parameters of the trace itself. However, as the "getdata" function shows, it does not return the actual value. Instead it subtracts the maximum rang value from it (for simplicity we have used 100). So if the depth is actually 64 meters the graph would be given the input of 64 - 100 = **-36**.

The layout variable is where, as the name implies, the layout of the chart is managed. We wanted the tracing value to decrease as it rose on the chart, since this looks more similar to a ship floating on the surface in relation to the seabed. As we did not find any good solution to invert the range from 0 on the top to 100 on the bottom and still fill in the lower part under the tracing with a color, we instead changed the graph to range from 0 to negative 100 and then subtracted 100 from the actual depth readings. From the example in the paragraph above, we get -36 as a plot on the graph which is 64 away from 100. By removing the values on the y-axis from the chart, we could create a new HTML element to place on top of the graph, with the visual appearance of an inverted range.

From the code in Listing 4.7 we would only get one value as this code alone would only update when the page is loaded and not give a real time view of the depth. Using Plotly's "relayout" function and placing it in an interval, we can pull inn new readings for the tracing at a given interval, as well as update the layout of the graph so previous tracings would get removed (Listing 4.8). In this case we chose the interval speed of 1000ms to not overload the touch panel's CPU.

To make the chart in the similar style as the other elements on the page we also had to make a function that extracts the coloring from the main document, as the chart is made in JavaScript and not in CSS. This is what the "getColor"-function does when the OpenBridge color variable is called into the function (Listing 4.8).

```
//updates y axis from input
var ctn = 0;
setInterval(function () {
  Plotly.extendTraces("chart", { y: [[getData()]] }, [0]);
  ctn++;
  if (ctn > 25) {
    Plotly.relayout("chart", {
      plot_bgcolor: getColor("--instrument-frame-color"),
      xaxis: {
        range: [ctn - 25, ctn],
        showgrid: false,
        showticklabels: false,
      },
      yaxis: {
        color: getColor("--element-active-color"),
        range: [-0, -101],
        showticklabels: false,
        gridcolor: getColor("--element-active-color"),
        dtick: 25,
      },
    });
    Plotly.restyle(
      "chart",
      {
        "line.color": getColor("--instrument-dynamic-color"),
        fillcolor: getColor("--instrument-track-color"),
      },
      [0]
    );
  }
}, 1000);
//Finds colors from the current theme
function getColor(color) {
  return getComputedStyle(document.documentElement).getPropertyValue(color);
}
```

Listing 4.8: Depth Graph updating

Self made icons

For our "Tanks & pumps" page, we wanted to have a visual representation of a pump next to its status. Since OpenBridge did not have any suitable icons, we had to make our own (Figure 4.8). Like all the components that complete the HMI, the self-made icon would need to follow the color theme to not look out of place when switching through the themes. To do this, we chose to use the CSS property: "filter". This does not change the color in the same way as the other implemented OpenBridge components, but by finding the color we wanted in the color variables, we could use online calculators to convert the RGB values to HEX (Dan's Tools, n.d.), and then convert HEX to filter values (Sonntag, n.d.).

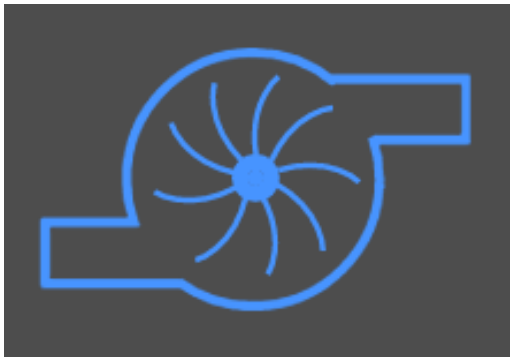

Figure 4.8: Pump icon

```
function customIcon(theme) {
  var x = document.getElementsByClassName("pump-img");
  var i;

  if (theme == "night") {
    for (i = 0; i < x.length; i++) {
      x[i].style.filter =
        "invert(12%) sepia(51%) saturate(7368%) hue-rotate(93deg) ...
           brightness(91%) contrast(103%)";
    }
  } else if (theme == "dusk") {
    for (i = 0; i < x.length; i++) {
      x[i].style.filter =
        "invert(57%) sepia(40%) saturate(5354%) hue-rotate(197deg) ...
           brightness(103%) contrast(103%)";
    }
  } else if (theme == "day") {
    for (i = 0; i < x.length; i++) {
      x[i].style.filter =
        "invert(46%) sepia(70%) saturate(5978%) hue-rotate(196deg) ...
           brightness(101%) contrast(102%)";
    }
  } else if (theme == "bright") {
    for (i = 0; i < x.length; i++) {
      x[i].style.filter =
        "invert(10%) sepia(85%) saturate(6056%) hue-rotate(197deg) ...
           brightness(103%) contrast(203%)";
    }
  }
}
```

Listing 4.9: Self made icons theme coloring

In essence, this function (Listing 4.9) will look through the HTML file in search for the elements that has the "pump-img" class bound to it. It will then apply the filter of the current theme to all these elements. For this to apply we called the "customIcon" function in the "activeTheme" function (Appendix A). So every time the theme is changed, the pump icon will also change color.

4.2 PLC programming

The programmable controller system was realized in e!COKCPIT by separating multiple Program Organization Units (POU) for the various simulated maritime systems. Each system uses functions for simulating data values towards a specific process. One of those control systems is "Engine_Control". Inside "Engine_Control" there are functions for simulating: effect, temperature, oil pressure and the thruster angle. Each of these function had to be manually controlled with the hardware we received from Wago (see Hardware Assembly 3.3).

4.2.1 Controller

Because we wanted to be able to regulate most functions with an analog value, rather than just a On/Off signal, we made the 8 digital inputs from the toggle switch (8-Channel Switch module 3.3.4) into a binary number system called "iBinary". Then we were able to separately regulate all functions from the same voltage regulator (Analog voltage regulator 3.3.5) simply by changing the binary value on the switch module corresponding to the wanted function, as shown in Listing 4.10. This listing shows how the different processes in "Engine_Control" are managed. When the digital input 7 is "TRUE", and digital input 8 is "FALSE", the digital inputs 1-6 acts as binary values. Since the input from the analog regulator gives a signal from 0 to 32760, we had to scale this down to the respected processes. For this we used the "FU2POINT" function to describe the linear equation between the signal and the process. "GVL.AzimuthPort" is the global variables for the "AzimuthPort" control system. "Effect", "Temperature", "Angle" and "OilPressure" are all monitored with the AnaView and AnaMon MTP functions.

```
rRegulator := FU2POINT(IoConfig_Globals_Mapping.iwRegulator,0,0,32760,100);

 IF DI7 AND NOT DI8 THEN
     LVL.rVoltage:=TO_REAL(TO_INT(rRegulator))/10;

      //Azimuth Thrusters
     IF LVL.iBinary= 1 THEN
        GVL.AzimuthPort_Effect:=FU2POINT(LVL.rVoltage,0,-30,10,100);
     ELSIF LVL.iBinary= 2 THEN
           GVL.AzimuthPort_Temp:=FU2POINT(LVL.rVoltage,0,0,10,200);
     ELSIF LVL.iBinary= 4 AND NOT GVL.autopilot THEN
           GVL.AzimuthPort_Angle:=FU2POINT(LVL.rVoltage,0,-35,10,35);
      ELSIF LVL.iBinary= 8 THEN
           GVL.AzimuthPort_OilPressure:=FU2POINT(LVL.rVoltage,0,0,10,100);
```

Listing 4.10: Simulating multiple analog values with a binary system

4.2.2 Object Oriented Programming

Planning a project for efficient, economical and structured solutions is a crucial aspect in the modern day automation. Structuring a PLC program to be reusable saves a lot of time, specially in industries where there are many repetitive activities (Engebretsen, 2017). This could be done by making programs into custom made function blocks, and is often referred to as Object Oriented Programming (OOP).

OOP is used for structuring a program to become a reusable part, for multiple similar processes. For some of our simulated processes, which we would use more than once, we managed to make our own OOP function-blocks as it would save us a lot of time, and add structure to the program. One example of such functions was the "Tank_Simulation" function-block which simulated heating and level control of a container, as shown in Figure 4.9. This was reused in four different tank systems, that we needed to simulate on the HMI.

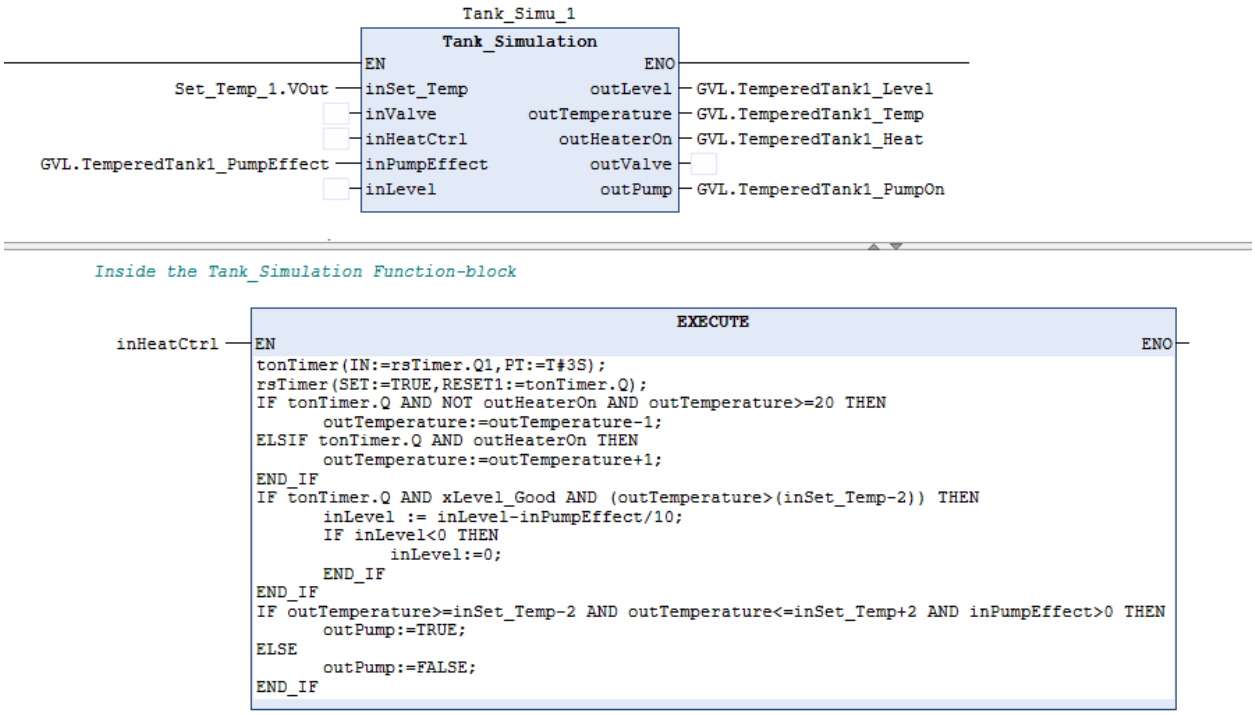

Figure 4.9: Tank_simulation

4.2.3 MTP functions

The monitored processes for "AzimuthPort" effect and temperature simulations, needed alarms to ensure that systems are running safely within acceptable limits, to avoid potential damage. We used the AnaMon function to oversee these limits as seen in Figure 4.10. Alarm high limit, Warning high limit and Alarm low limit is enabled to ensure that the user is informed if a systems exceeds a limit.

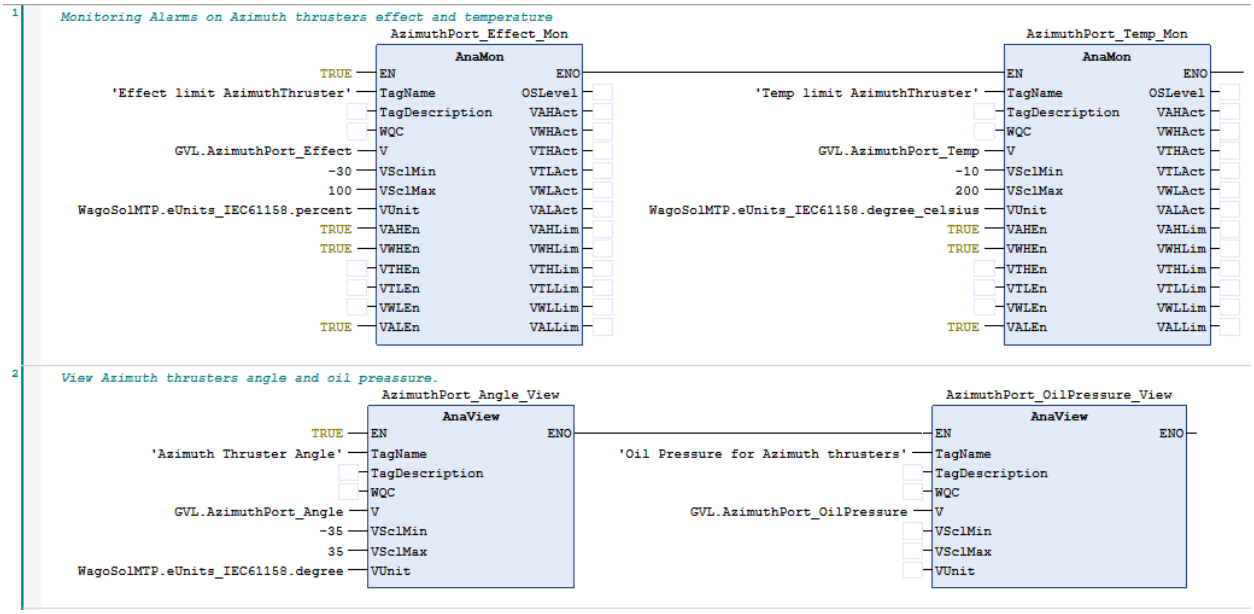

Figure 4.10: MTP - Function Blocks

Our ambition for the HMI were to have a highly interactive interface. Processes such as light control were intended to be controlled from the HMI. For that we required other functions than the previous indicator elements like AnaView, which only monitor values from the PLC. Instead we needed functions that monitor the return values from the HMI. Such functions are MTP Operation Elements and they are designed for transmitting values from the POL into the PLC (see MTP 2.3). We found the two operator elements BinMan and AnaMan useful for a interactive interface. BinMan is used to set or reset a binary value in the PLC from the POL. We used this for turning lights On/Off in the PLC from the HMI. We also used AnaMan to change the angle on the lights. AnaMan returns a output value from the POL that checks the limit on the input.

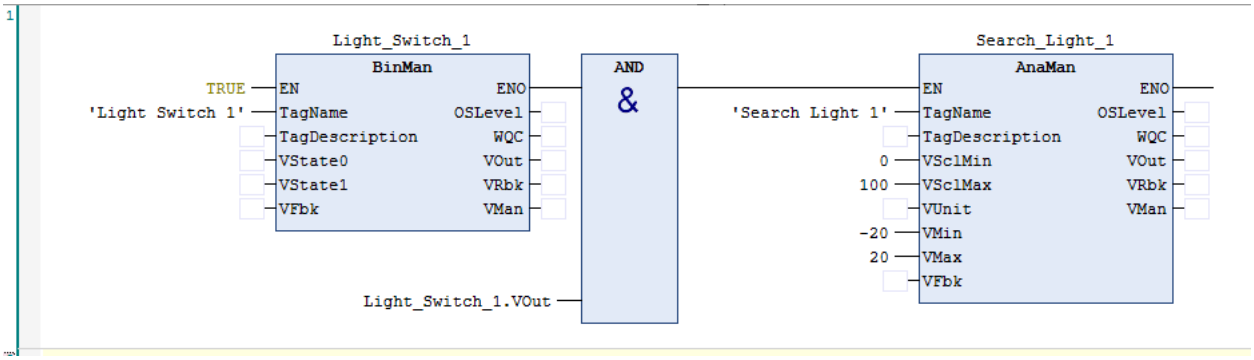

Figure 4.11: Search light

4.2.4 OPC-UA communication

After the simulated maritime systems were created we had to make the variables we wanted to send and receive into global variables. This made it easier to identify the variables we wanted to send via OPC-UA. Lastly, we would need to make the variables available for transfer via OPC-UA. This was done by creating a "Symbol Configuration" in e!COCKPIT. A Symbol Configuration is created within the application for our PFC200 in e!COCKPIT. The Symbol Configuration must enable the support for OPC-UA features. Inside the Symbol Configuration was where we put all the variables which would be transferred to the HMI.

4.3 OPC UA connection

The OPC UA connection was established using the PFC200 (see PFC200 3.3.1) as a server, and coding a client using the NodeJS-framework (see NodeJS 2.5). The PLC uses OPC-UA as the standard communication protocol and acts as the server in the communication. To enable OPC-UA on the PLC, we needed to change a few settings.

4.3.1 Server Side

Firstly, we needed to access the PLCs Web-based Managment (Figure 4.12), by entering the IP-address of the PLC in a web-browser. After accessing the settings, we needed to check if OPC-UA is enabled, by going to "Fieldbus->OPC UA->OPC-UA" and enable the service.
Then we had to disable Port Authentication, as we would be using a open connection in this project. We could do this by going to "Configuration->Ports and Services->PLC Runtime Services->e!RUNTIME->Port Authentication" and turn it off.
The OPC-UA server was now ready to transfer variables from the SymbolConfiguration created in e!COCKPIT, to a client

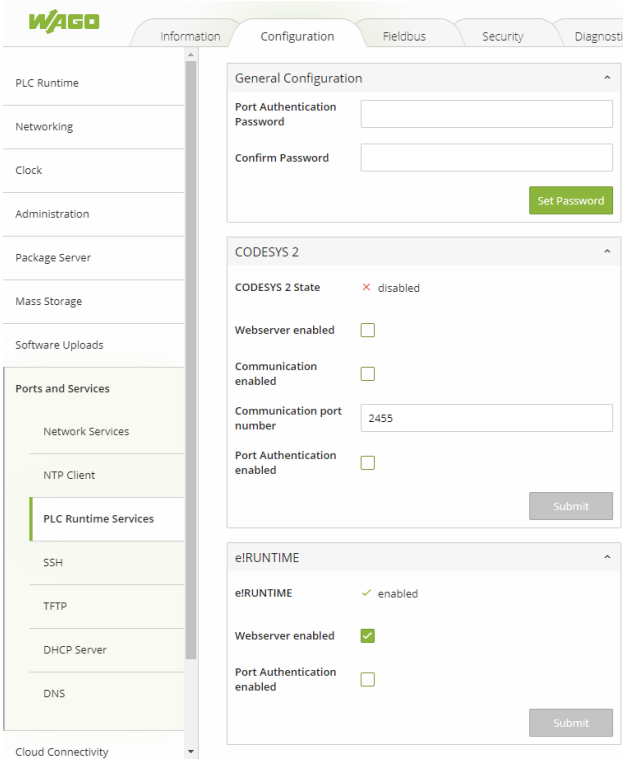

Figure 4.12: Wago Web Based Management (Wago, n.d.-d)

4.3.2 Client Side

The OPC-UA client needed to be coded by hand with Node.js, which made it more of a difficult task than the server. There were a simple client on the node-opcua GitHub which we used as inspiration (Rossignon, n.d.-b). Firstly, we needed to make a new JavaScript-file and include the "node-opcua" module. We could then create a variable called "client" and connect it to the PLC's IP through the "connect"-function. We could then create a session by using the "session"-function.

```
const opcua = require("node-opcua");

const options = {
  endpoint_must_exist: false,
  keepSessionAlive: true,
  allowAnonymous: true,
};
const client = opcua.OPCUAClient.create(options);
    //endpointUrl = "opc.tcp://'PLC-IP':4840"
```

```
var endpointUrl = "opc.tcp://192.168.10.184:4840";

async function main() {
  try {
    // step 1 : connect to
    console.log("trying to connect to " + endpointUrl);
    await client.connect(endpointUrl);
    console.log(" connected to ", endpointUrl);

       // step 2 : createSession
    const session = await client.createSession();
    console.log("session created !");
```

Listing 4.11: Getting a connection with Node-opcua

When the PLC server was set up correctly, running the script above (Listing 4.11) would now log "connected to 192.168.x.x:4840" and "session created". This meant we had established a connection to the PLC and that a session was successfully created. The next step was to set up a subscription so all our variables could be monitored in real time. We chose to have one subscription for all variables as we only had a handful of variables that was going to be monitored, and we wanted to keep it as simple as possible.

Next, we needed to install the monitored items, and this was were we required the address of the variables from the PLC. We made a function called "MonitorItem(id)" where "id" was the address of a variable. Then we used the node-opcua module function "on("changed")" which detects change in a subscribed variable, and executes a command afterwards. We wanted this command to send the changed variable's name and value as an array to the HMI. As the OPC-UA client will be a "child-process" of Web.js (Figure 3.15) we can use the function "process.send()" (Listing 4.12).

```
//Install Monitored Variable
async function MonitorItem(id) {
....
....
  //Read changed variable and send to next link
      monitoredItem.on("changed", (dataValue) => {
        var msg = [msgSend, Math.floor(dataValue.value.value*10)/10];
        console.log(" value has changed for: ",msg[0], msg[1]);
        process.send(msg);
      });
}
```

Listing 4.12: Function detecting a change of variable from the PLC

Now that the functions was in place, we just had to insert the addresses of the variables. This proved more difficult than expected, as there were no way to find the address directly from the

PLC. After some research, we discovered we could obtain the address from the Ignition software (Inductive Automation, n.d.). Nevertheless, there were some issues, as the address we got from ignition did not seem to work either.

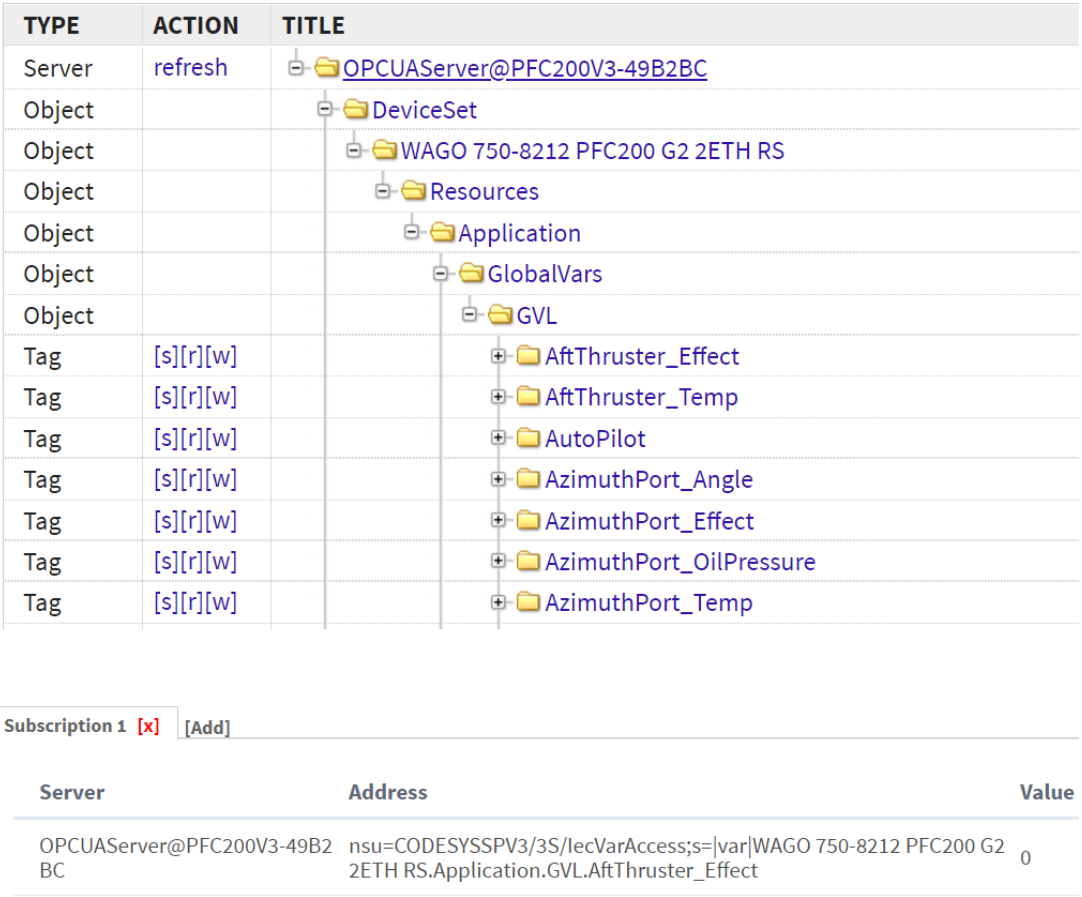

Figure 4.13: Screenshot of the Ignition OPC-UA client (Inductive Automation, n.d.)

We tried putting the address from Ignition into the client (Figure 4.13), but only received errors of invalid variables. That is when we noticed a small difference in our Ignition address and the one from the tutorial (Braun, 2016). Our address started with "nsu" (namespace URI), while the one from the tutorial began with "ns" (namespace index). This small change in the address was most likely due to an update that had taken place somewhere in the last 5 years from the tutorial was made(2016) and is shown in the listing below (Listing 4.13).

```
//Address from our Ignition client
"nsu=CODESYSSPV3/3S/IecVarAccess;s=|var|WAGO 750-8212 PFC200 G2 2ETH RS..."
//Address from the tutorial
"ns=4;s=|var|WAGO 750-8212 PFC200 G2 2ETH RS..."
```

Listing 4.13: Difference in address in Ignition

When we now tried to input the address in our client, we noticed the variables was detected and monitored. By inputting all the variables we needed to monitor in the MonitorItem(id)-function, the OPC-UA client could now monitor all changed variables (Listing 4.14).

```
//INPUT ALL ITEMS FOR MONITORING
    MonitorItem("ns=4;s=|var|WAGO 750-8212 PFC200 G2 2ETH ...
        RS.Application.GVL.iDirection");
    MonitorItem("ns=4;s=|var|WAGO 750-8212 PFC200 G2 2ETH ...
        RS.Application.GVL.iActualDir");
    ....
    ....
    MonitorItem("ns=4;s=|var|WAGO 750-8212 PFC200 G2 2ETH ...
        RS.Application.GVL.rCurrent");
```

Listing 4.14: Monitoring variables from the PLC

Since we did not want the whole address to get forwarded to the HMI, we used the "substring" function to remove the first part of the address, thereby only forwarding the name of the variable. By logging the changes of variables, opening the console would give the following results (see Figure 4.14) indicating that the client was working.

Figure 4.14: Console log of OPC-UA client

4.4 WEB.JS

In order to transfer the variables from the OPC-UA client to the HMI, we required a intermediary which could both host and communicate with the website, and also connect and communicate with the OPC-UA client. This would be our main script called "web.js" (Figure 3.15). As previously mentioned in NodeJS 3.5.1, we used node-modules' "socket.io" for the website and "child_process" for the OPCUA-client.

4.4.1 Locally hosting a website

To host a local website that could utilize socket communication, we were required to open a port for http-requests using the http-module (OpenJS Foundation, n.d.-d). We then pointed to our projects root folder and rendered the "index.html" file using the express-module (ExpressJS, n.d.). This ensured we had a website running that could make use of socket communication (Listing 4.15).

```
const express = require('express');
const app = express();
const http = require('http').Server(app);
const io = require('socket.io')(http);
const { fork } = require('child_process')

const PORT = 9999;

//Listens to port 9999
http.listen(PORT, () => {
    console.log('Server running at http://localhost:${PORT}/');
});

//Points to root directive
app.use(express.static(__dirname + '/'));

//Renders index.html
app.get('/index', function(req, res) {
    res.render(__dirname + "index.html");
});
```

Listing 4.15: Code showing the backend of a webpage running on "localhost:9999"

4.4.2 Forwarding values from PLC to HMI

When creating a communication between the HMI and OPC-UA client, there were a lot to keep in mind. As we were dealing with both socket communication and parent-child communication at once, creating a interaction between the two tended to get a little complicated.

Initiating the OPC-UA client

Firstly, we needed to initiate the OPCUA-client from the "web.js" script. We chose to initiate the script only when the HMI/website was in use by using the function **io.on("connection"...)**. This gave us the advantage that if the opc-ua client crashed or there were errors, we could just refresh the web-page and the client would initiate again (Listing 4.16).

```
io.on('connection', (socket) => {
    const n = fork(`./opc.js`);
    ...
}
```

Listing 4.16: Initiating the OPC-UA client

Sending from PLC to HMI

As shown in Listing 4.12, the OPC-UA client sends the variables from the PLC in an array containing the name and value, in that order. The "web.js" script receives these values and forwards them to the HTML/HMI using Socket.IO and Child_Process (3.5.2) as shown in Listing 4.17.

```
io.on('connection', (socket) => {
    const n = fork(`./opc.js`);
    n.on('message', (m) => {
        io.emit('PLC', m);
    });
    ...
}
```

Listing 4.17: Forwarding values from PLC to HMI

Receiving the variables in the HTML

When using socket.io in HTML, we had to include the "socket.io.js" script as an external JavaScript. We then called the functions of this script and put them in a variable "socket". As explained in Socket.IO and Child_Process 3.5.2, we could use "socket.on" to receive messages from the server. These messages, containing the name and value of a variable, would then be placed in a container with correlating identity as the name of the variable. The value would then be placed in the "innerText" of the container as shown in Listing 4.18.

```
<body>
    ...
    <div id="GVL.AzimuthPortEffect"> 72.3 </div>
    ...
</body>
<script>
    src="/socket.io/socket.io.js";
    var socket = io();
    socket.on('PLC', function(msg) {
        document.getElementById(msg[0]).innerText=msg[1];
    }

</script>
```

Listing 4.18: HTML receiving variables and inserting them in the corresponding -container

An array "msg" is received from the server, as shown in Listing 4.18 above. One example of this is when the array contains the string "GVL.AzimuthPortEffect" in msg[0] and the value "72.3" in msg[1]. By using the function "document.getElementByID()" we find a -container with corresponding identity to the array string. The value of the array replaces all previous contained text in the by using "innerText", displaying the value on the website.

4.4.3 Values.js

The complication of using id's when receiving variables, was that the id must be unique, meaning it can only be used once. When the variable from the PLC was going to be used on several containers, we had to perform some extra steps. We created the file "Values.js" to keep track on all variables that would require to be used more than once. After some testing, we decided to loop the script at a 100 millisecond interval, as it could update variables frequently enough that there would not be any stuttering on the HMI, but at the same time not use too much processing power. We could then use the JavaScript-function "getElementByID()" to extract values, names and attributes from a container. This way we could get the value from one container and put it in another container. For the OpenBridge components we needed to set the predefined attributes to change visuals, like the compass or thrusters. If we just wanted to use the value as a string on another container, we could use the "innerText" function as demonstrated in Listing 4.19 below.

```
//Changing the compass needle based on the value of "GVL.iCurrentDir"
  currentDir.setAttribute("style", "transform:rotate...
    ("+document.getElementById("GVL.iCurrentDir").innerText +  "deg)");
//Copying the value of Azimuth Port and putting it in Azimuth Starboard
  document.getElementById("AzimuthPort_Effect").innerText=...
    document.getElementById("GVL.AzimuthStarboard_Effect").innerText;
```

Listing 4.19: Values.js: Showing how we could use the same value for different components

4.4.4 Sending values from the HMI to the PLC

Forwarding changes from the HMI to the PLC was a more difficult task than the other way around. The reasoning for this was that when sending data from the PLC to the HMI, the HTML would only look at the value as a string, no matter what data-value it actually was. When sending from the HMI to the PLC however, we found that the data-value had to be specified or else the PLC will not accept it. This meant that we could not have a unified function for sending variables from the HMI, but instead required one function per data-value. For this project we used three data-values; float/real, integer, and boolean.

HTML to Web.js

As we had three data-values to deal with, we created three simple functions in the HTML script as shown in Listing 4.21: SendBool(), SendFloat() and SendInt(). These functions received the name and value of the variable which called the given function (Listing 4.20). The task of these function was to distinguish the data-values from each other by sending a message to the server with an identifier corresponding to the data-value. The server could then execute different commands depending on the identifier.

```
<div name="GVL.AutoPilot" onclick="SendBool(this.attributes['name'].value,...
this.classList.contains('active'))">
```

Listing 4.20: A button getting pressed will call the function "SendBool(name,value)"

```
function SendBool(name,value){
      console.log(name,value);
      socket.emit('bool', [name,value]);
    }
    function SendFloat(name,value){
      console.log(name,value);
      socket.emit('float', [name,Number(value)]);
    }
    function SendInt(name,value){
      console.log(name,value);
      socket.emit('int', [name,Number(value)]);
    }
```

Listing 4.21: The different HTML functions for sending messages based on data-type

Web.js to OPC-UA client

When "web.js" received the message it would assign a data-type to the incoming value, based on the identifier of the message (Rossignon, n.d.-a). This value would then be forwarded again to the OPC-UA client as shown in Listing 4.22 below.

```
    socket.on('bool', msg => {
        dataToWrite = {
            dataType: 1, //bool
            value: msg[1]
        }
        n.send([msg[0],dataToWrite]);
    });
```

Listing 4.22: Web.js: One of the three functions receiving a value and assigning datatype "1" representing a Boolean value

OPC-UA client to PLC

At the moment the OPC-UA client received a message from the HMI, it executed the function "SendPLC()" which added the OPC-UA address to the variable name. The message would then contain the value with correct data-type assigned, in addition to the complete OPC-UA address, which allowed us to use the node-opcua function "WriteSingleNode()" (Rossignon, n.d.-b) which overwrites the PLCs variable with the message variable (Listing 4.23).

```
//ON MESSAGE
process.on("message", (m) => {
  SendPLC(m[0], m[1]);
});

//Write function
async function SendPLC(id, value) {
  await session.writeSingleNode(
    "ns=4;s=|var|WAGO 750-8212 PFC200 G2 2ETH RS.Application." + id,
    value
  );
}
```

Listing 4.23: Opc.js: The OPC-UA client receiving the message from "web.js", adding the address to the name, and writing the value to the PLC

4.5 Finalizing the project

After everything was programmed and functional, the last step was to put everything on the TP-600 (Touch Panel 600 3.3.3) so it could run independently without a computer hooked up to it. This was were Docker (Docker 3.6.1) came in to play, as most WAGO hardware like PLCs and touch-panels support Docker and it was one of our aims for this project (Project Aims and Objectives 1.2). This meant that we could download Docker onto the TP-600 and run an image containing the project.

4.5.1 Building a docker image

Before building a docker image from the folder containing the project, we required a so-called "Dockerfile" which contains the data for this particular image (Listing 4.24). We followed a guide on Nodejs.org (OpenJS Foundation, n.d.-c), and edited the parts we needed, one of which being that the TP 600 processor runs on ARM32 architecture (Wago, n.d.-c), meaning we required the arm32 node base image (NodeJS Docker Team, n.d.). We then exposed port 9999 for convenience and added the command "web.js" which would launch the project. Finally, we ran the command "**docker build . -t orjpet/hmi**" inside the project folder to build our docker image.

```
FROM arm32v7/node:15.12.0-alpine3.10
RUN apk update \
    && apk add openssl
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 9999
CMD [ "node", "web.js" , "/bin/sh"]
```

Listing 4.24: Dockerfile

4.5.2 Getting the image on the TP-600

To download the image on the TP 600, we first needed to install docker. We used WAGO's GitHub (WAGO Norge AS, n.d.) for this as it had a step by step guide. We used PuTTY (PuTTY 3.6.2) to access the TP-600's terminal, where we had access to the inner workings of the panel. After having installed Docker, we pulled our image from the cloud by using the command "**docker pull orjpet/hmi**". Then we ran the image as a container by using the command "**docker run -d –restart always -p 9999:9999 –name hmi orjpet/hmi**", which created a container with the name "hmi" and access to port 9999. By using "**docker start hmi**", we could initiate the container which would start the project. The HMI was then up and running and accessible on "localhost:9999", which we configured to be our homepage on the touch panel. When the website was visible on the TP 600 we could see that the PLC-variables where present and changing, meaning everything was working as it should.

Chapter 5

Results

In this chapter we will show the results from our bachelor's project. We will explain the functionalities of each HMI page, as well as how to simulate the variables from the PLC by using switches and an analog voltage regulator.

5.1 Top Navigation Bar

The horizontal bar at the top of the website is called the "Top Navigation Bar" (Figure 5.1). It follows the user to every web page, and lets the user navigate to every part of the website. The bar contains five different buttons that each opens different applications.

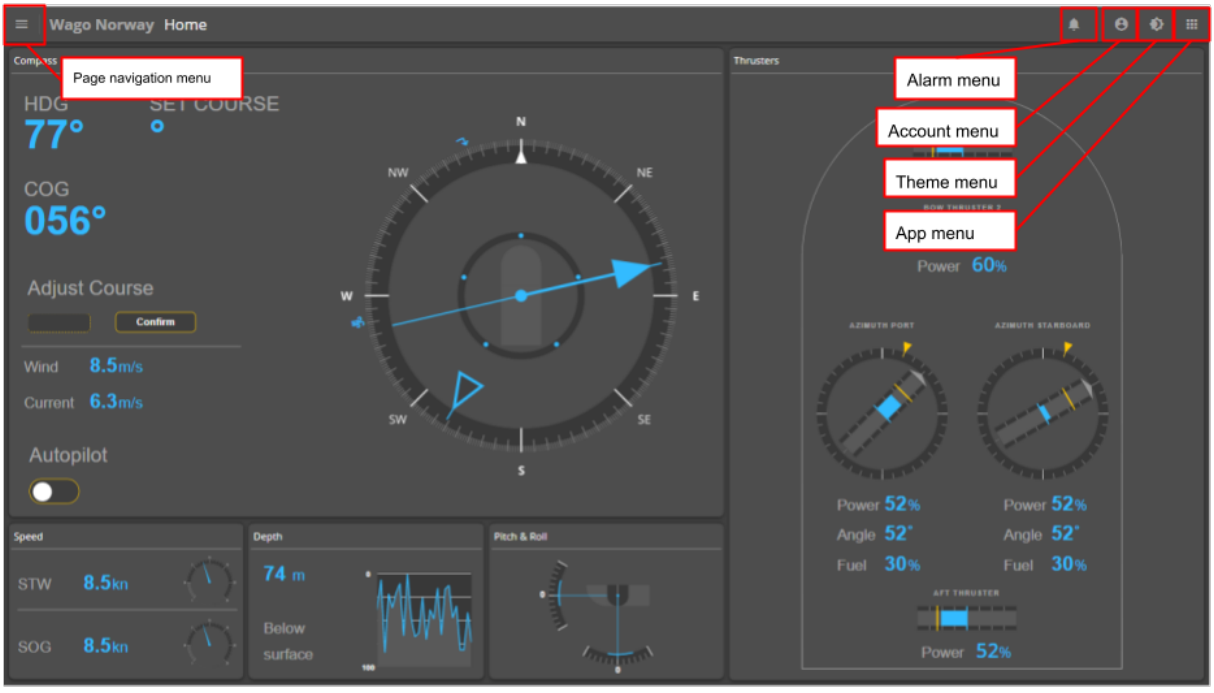

Figure 5.1: Top Navigation Bar

5.1.1 Page Navigation Menu

The page navigation menu (Figure 5.2) is where the user can choose between, and access, all the different pages on the website. The menu pops up when the "page navigation button" in the upper left corner is clicked or touched. The menu disappears when one of the page buttons on the menu is clicked, taking the user to the chosen web page, or when the "page navigation button" is clicked again.

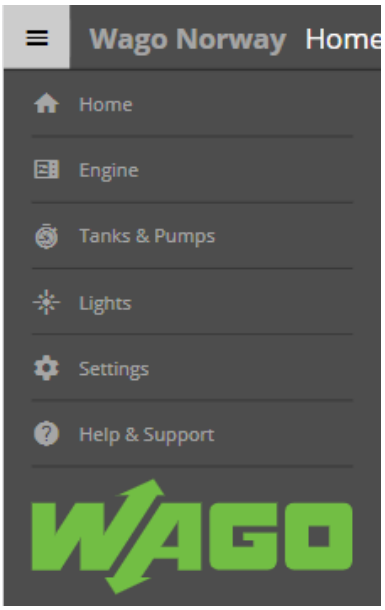

Figure 5.2: Result: Navigation Menu

5.1.2 Alarm list

The alarm list gives the user a clean overview of the alerts coming from the connected PLC system. When the HMI receives an alarm, a number will be added in the upper right corner of the "bell-icon", as shown in Figure 5.3. If there are three alarms, the number "3" would appear on the bell-icon. In the alarm section, the user is able to acknowledge the alarms by pressing "Ack", making the alarm disappear from the list.

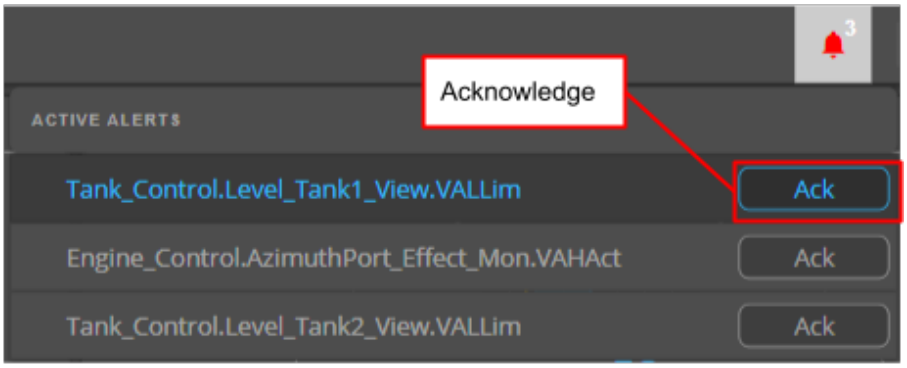

Figure 5.3: Alarm list

5.1.3 Account login

The account login section (Figure 5.4) serves only as a visual illustration of how the account login could look like when using the OpenBridge design. It has no functionality other than letting the user type in letters in the input section.

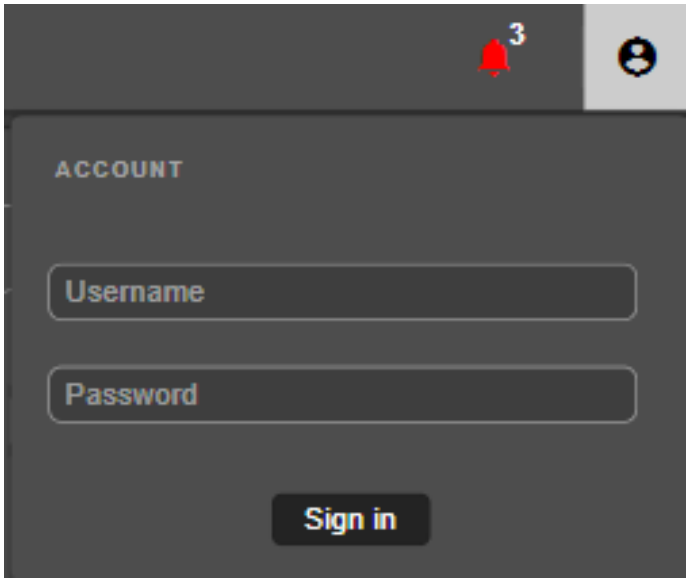

Figure 5.4: Account login

5.1.4 Theme color menu

The theme color menu allows the user to change the theme color of the website. There are 4 different themes to choose from; "Night", "Dusk", "Day" and "Bright". The user can change the theme by clicking on one of the icons shown in Figure 5.5. The theme will be set for all the pages on the website and will stay on the chosen theme until another is chosen or until the website is reloaded.

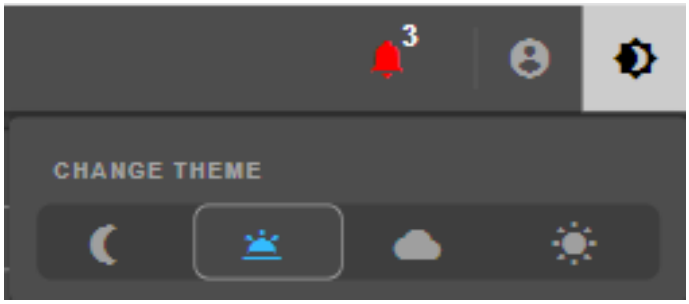

Figure 5.5: Theme color menu

Website themes

The **day** theme displays the website in a light to white primary color, and blue as the secondary color. This theme is great for highlighting the details on the website when there is a lot of light in the room.
The **night** theme displays the website in a very dark to black primary color, with green and yellow as secondary colors. This makes it more comfortable for the eyes if the screen is in a very dark room.
The **bright** theme displays the website in a very bright white primary color, and dark blue as secondary color, making the system visualizations easy to see when the room is very bright.
The **dusk** theme displays the website in a light dark color that is very appealing to the user in most conditions. The secondary color is a pleasant clear blue color that also highlights all the important systems. The four themes can be seen in Figure 5.6.

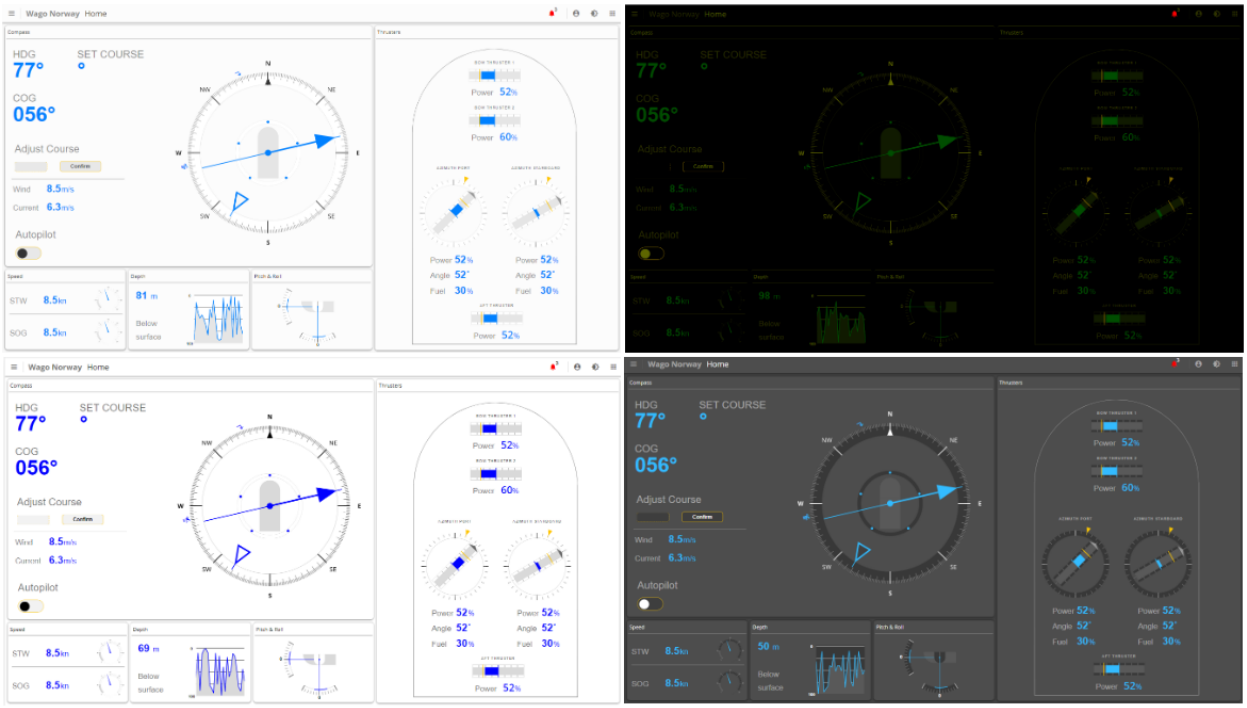

Figure 5.6: Website themes

5.1.5 Application menu

The application menu (Figure 5.7) is only a visual illustration of how the application menu would look like with the OpenBridge design. It displays a list of different icons with a satisfying blue hover color. The icons are meant to serve as different applications in a potentially functional version.

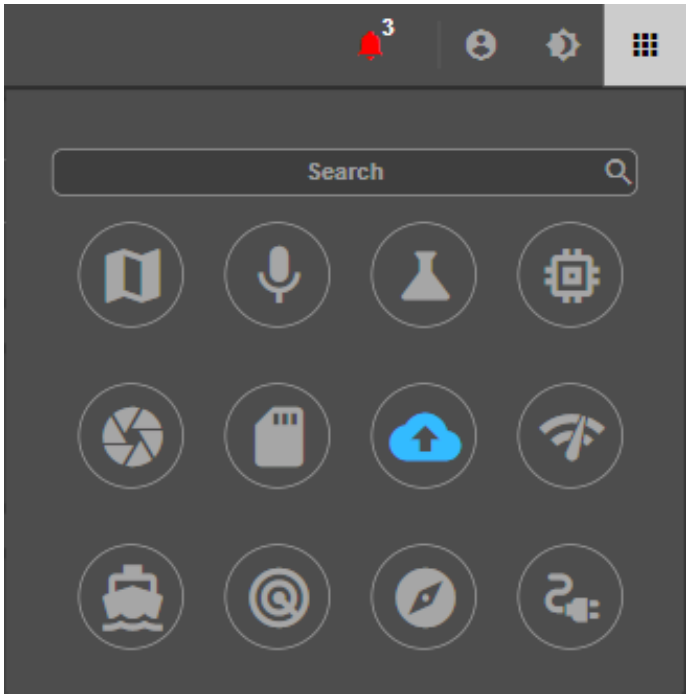

Figure 5.7: Application menu

5.2 Home page

To easily show how the HMI works, we have framed and numbered the different components and sections on each page in Figure 5.8, with an explanation below, referenced with the corresponding number. This concept applies on the "Engine page" 5.3, "Tanks and Pumps page" 5.4 and "Lights page" 5.5 as well.

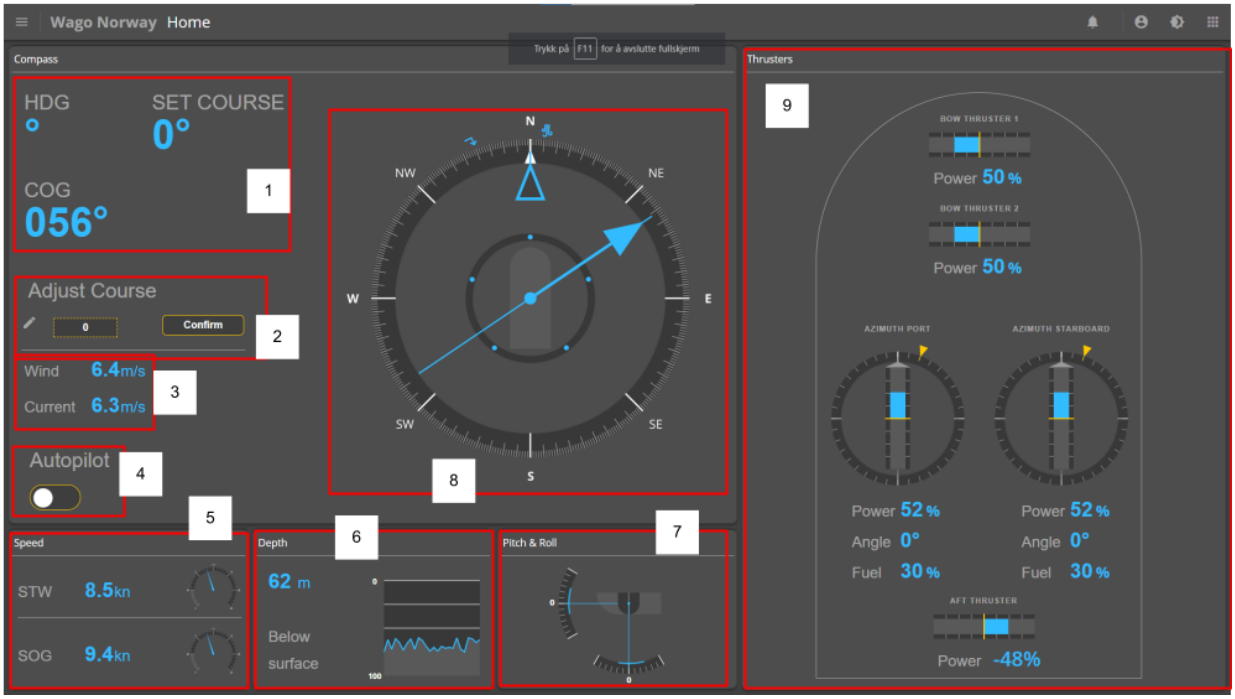

Figure 5.8: Home page

1. The **navigation values** is where you monitor the numbered values of the course over ground (**COG**), the heading of the ship (**HDG**) and the (**Set Course**) that the autopilot will follow when active. From the **COG** you would get the accurate information on which direction the ship is moving, but also the direction it is pointing from the **HDG**, as that can differ in relation to the water current and wind.

2. **Adjust course** is where you would adjust the **Set Course** either before or after you turn on the autopilot, by typing in a course in degrees and pressing confirm.

3. The **wind and current display** will display their respective values in meters per second.

4. The **autopilot button** toggle the autopilot ON or OFF. This example shows the OFF state.

5. The **speed overview** shows the speed through water (**STW**) and the speed over ground (**SOG**). From this you will be able to see if and how much the waters current affect the speed of the ship.

6. The **depth overview** is a real time visualization of the seabed in relation to the surface. This is a good way to monitor the seabed and watch for any sudden changes or gradually moving into shallow waters.

7. **Pitch & Roll:** As of now the pitch and roll is not functional and only serve as a visual representation of an element that can be use in that position. The choice of leaving this as a

nonfunctional element, was due to time management and prioritizing of the elements that is easier to demonstrate with the analog potentiometer and binary switches. Since these where the demonstration instruments we was given to us for simulating values.

8. The **compass** is a combined visualization of five different elements. The compass needle show the **COG**. The hollow blue triangle at north position indicates the **Set Course**. At position north-northwest there is 90 degree arrow that indicates the direction of the current and slightly right of north there is a symbol the indicates the wind direction. In the center of the compass there is a very simplified boat that is visualizing the **HDG** in relation to the **COG**.

9. The **thrusters overview** is an overview of the power usage of all the thrusters on the ship, with some additional information on the azimuth thrusters. Form this view you can easily see how much and in which direction the thrusters are powered by looking at the blue bars on each visualization. You also have the percentage values to more accurately know the power output. Since the azimuth thrusters is the main thrusters, you also have the propellers angel and fuel level displayed.

5.3 Engine page

The engine page (Figure 5.9) displays all the important parts that has to do with the engine, such as fuel level, temperature and power usage. This page has no buttons to interact with, but displays a lot of live updated values that may be important to the user.

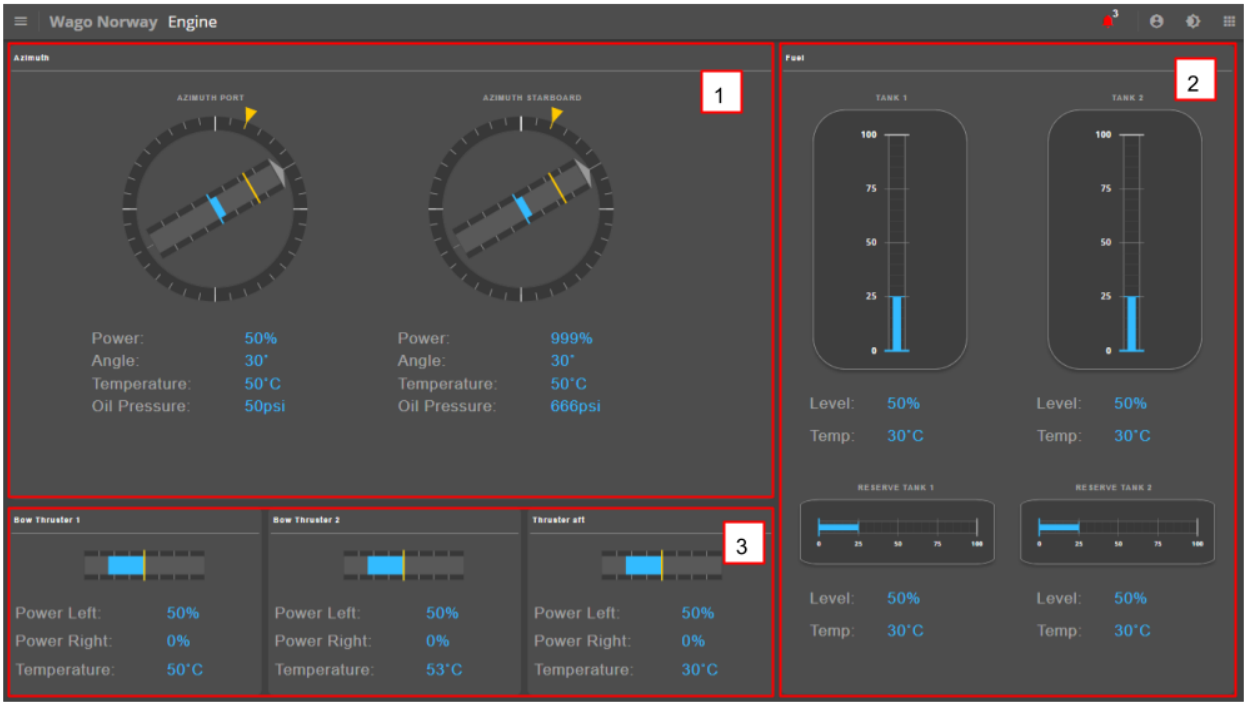

Figure 5.9: Engine page

1. The **azimuth** section displays a visualization of both the azimuth engines, including their power usage, angle, temperature and oil pressure. Power usage is measured in percent, angle in degrees, temperature in degree Celsius, and oil pressure in psi (pounds per square inch).

2. The **fuel** section displays visualizations of the two main fuel tanks of the ship, and the two reserve fuel tanks. Each of the visualizations include a measure of fuel level in percent and temperature in degree Celsius.

3. The **thrusters** section is divided into three parts; two bow thrusters and one aft thruster. Each of them visualizes the direction of power in percent, and thruster temperature in degree Celsius.

5.4 Tanks and Pumps page

The tanks and pumps page is divided into four tank sections and four pump sections, where each of them displays a simulation of a tank, heater and values, with its corresponding pump. The example in Figure 5.10 shows each of the sections divided into different parts.

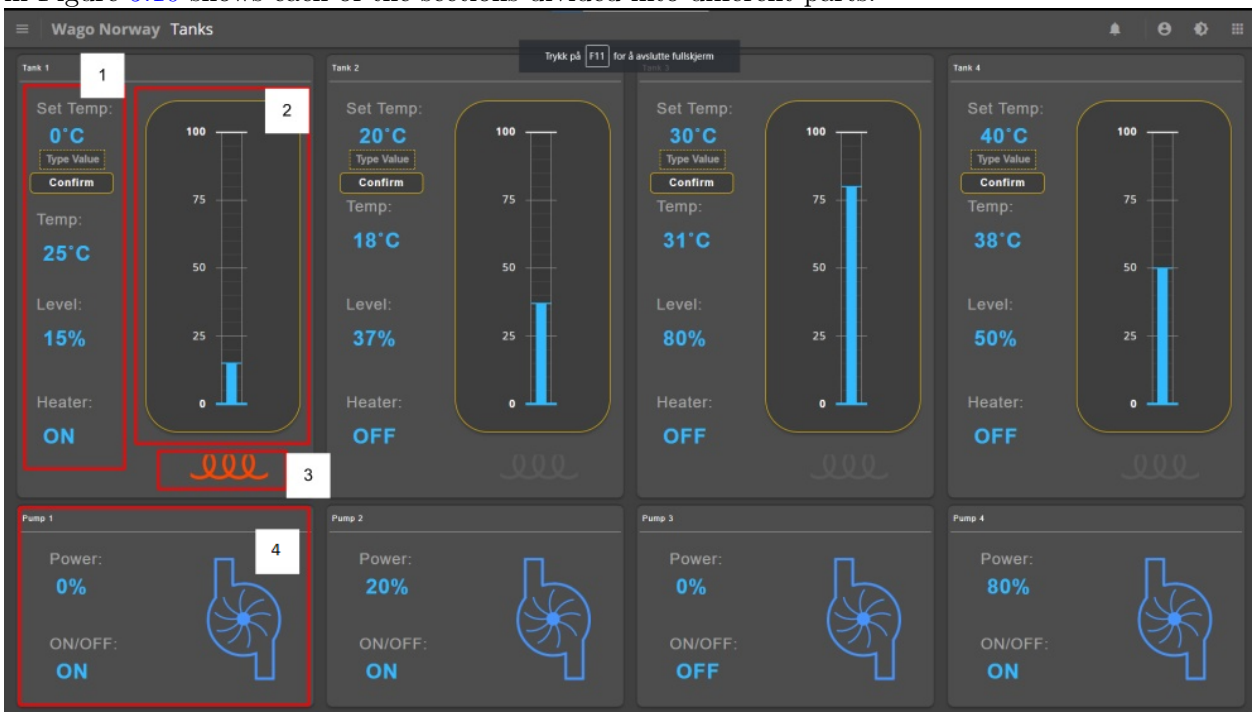

Figure 5.10: Tanks and Pumps page

1. The **values** section is where all the measurements of the tanks is displayed, but it also includes a input where the user can enter a desired temperature the tank should hold. The values displayed are the set temperature in degrees Celsius, level in percent, and temperature in degrees Celsius.

2. The **tank** section is a visualization of a tank that shows how much content is left inside.

3. The **heater** section is a visualization of a heater that is displayed with a red/orange color when warm, and a grey color when off.

4. The **pumps** section displays the power of the pump in percent as well as a illustration of the pump and whether it is turned on or off.

5.5 Lights page

The lights page (Figure 5.11) acts as an interactive control panel for the exterior spotlights on the ship with adjustabilities such as angle and brightness.

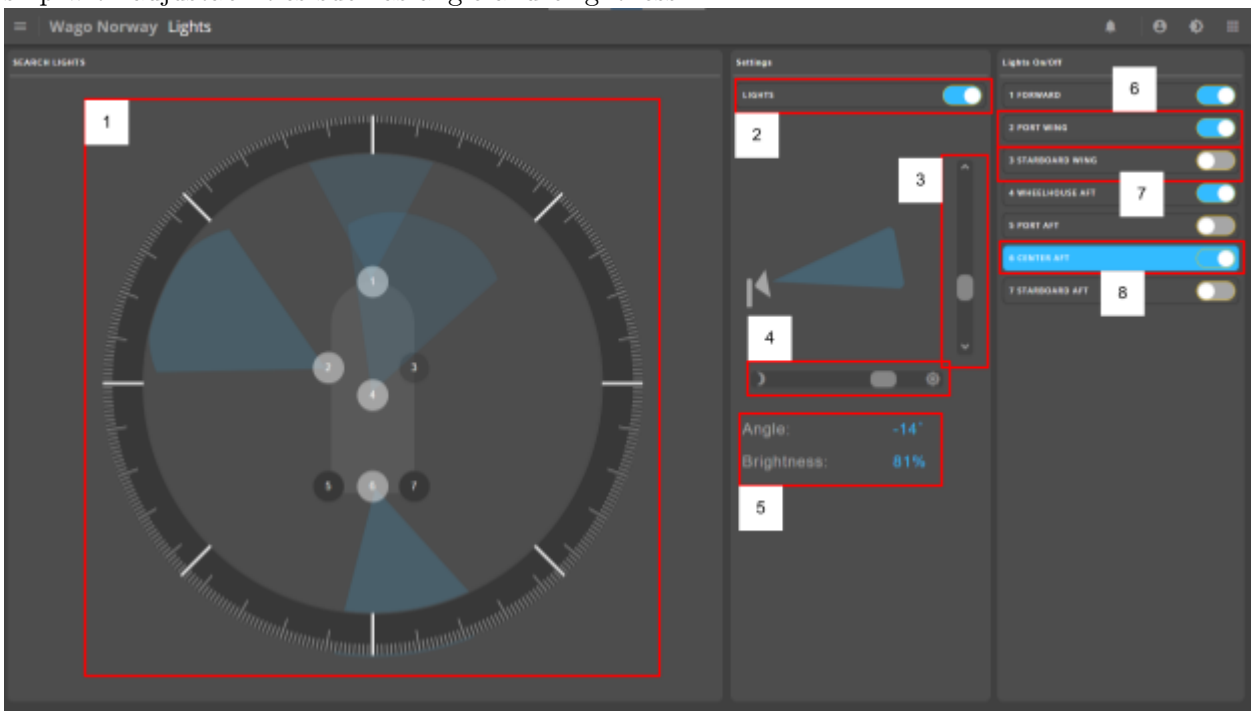

Figure 5.11: Lights page

1. The **lights visualization** is where the graphic status of all the spotlights are displayed. It shows the direction and brightness by the light blue areas and the opacity of these.
2. The **lights switch** is the main ON or OFF switch for all the lights. This button is not functional at this moment, but is placed there as a concept that can be implemented in the future.
3. The **angle slider** is where you can change the angle of the selected light from -20 to 20 degrees by dragging the slider up and down.
4. The **brightness slider** is where you can change the brightness of the selected light in a percentage fashion from 0 to 100.
5. The **light values** is displaying the numbered brightness and angle values of the selected light.
6. This is a representation of a light that is turned ON with the toggle button in the right position
7. This is a representation of a light that is turned OFF with the toggle button in the left position
8. This is a representation of the selected light that is turned ON and is ready to be modified by the sliders, represented as 3 and 4.

5.6 Settings and Support pages

The settings and Support pages (Figure 5.12) are only visual presentations of what these pages would look like when using the OpenBridge design. These pages have no functionality other than

styling effects on the cards.

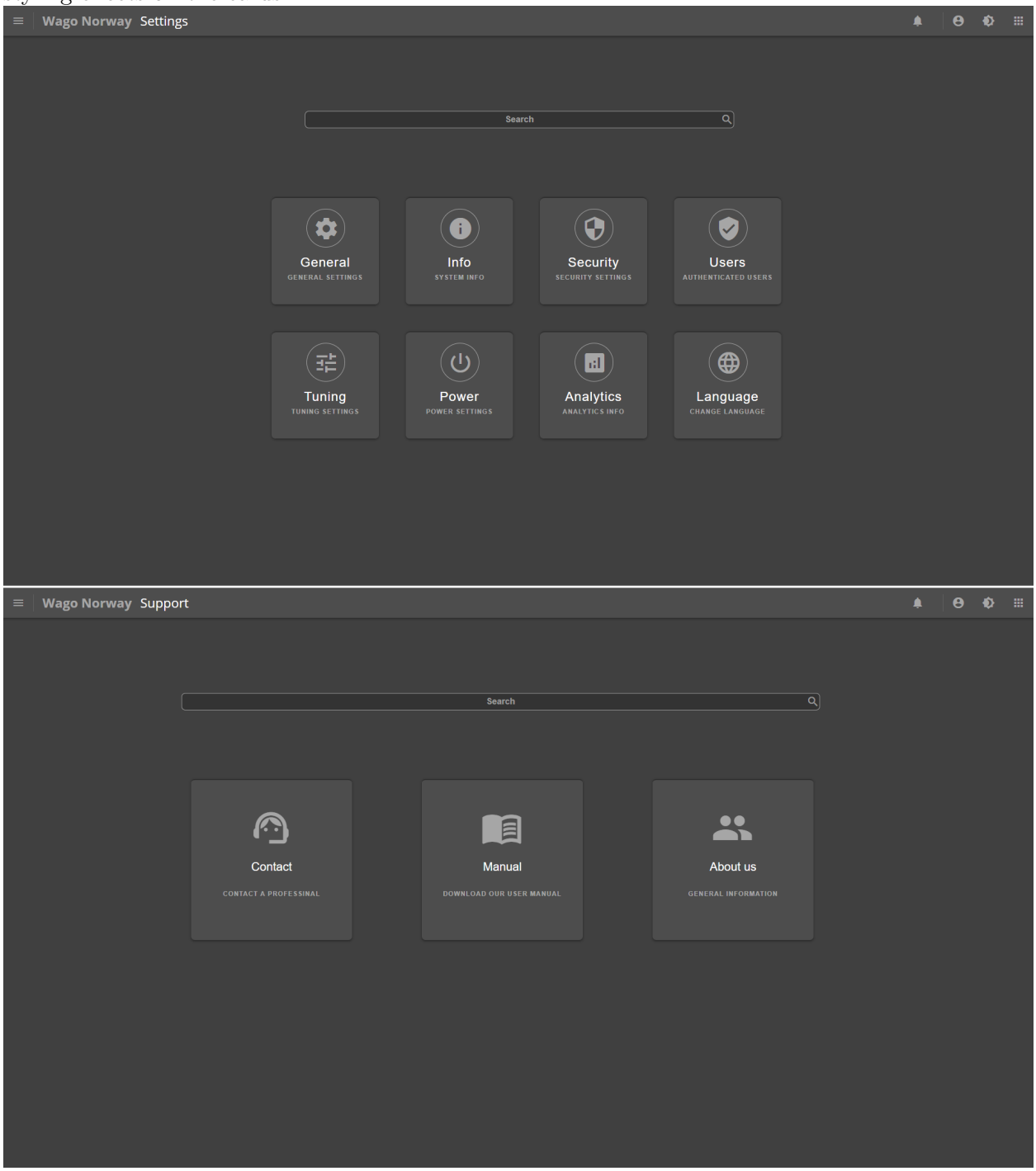

Figure 5.12: Settings and Support pages

5.7 Control system

With the test equipment for the PFC 200, the user can control multiple different processes. The control system is a composition of the Analog voltage regulator (Figure 3.11) and the 8-channel Switch interface (Figure 3.9).

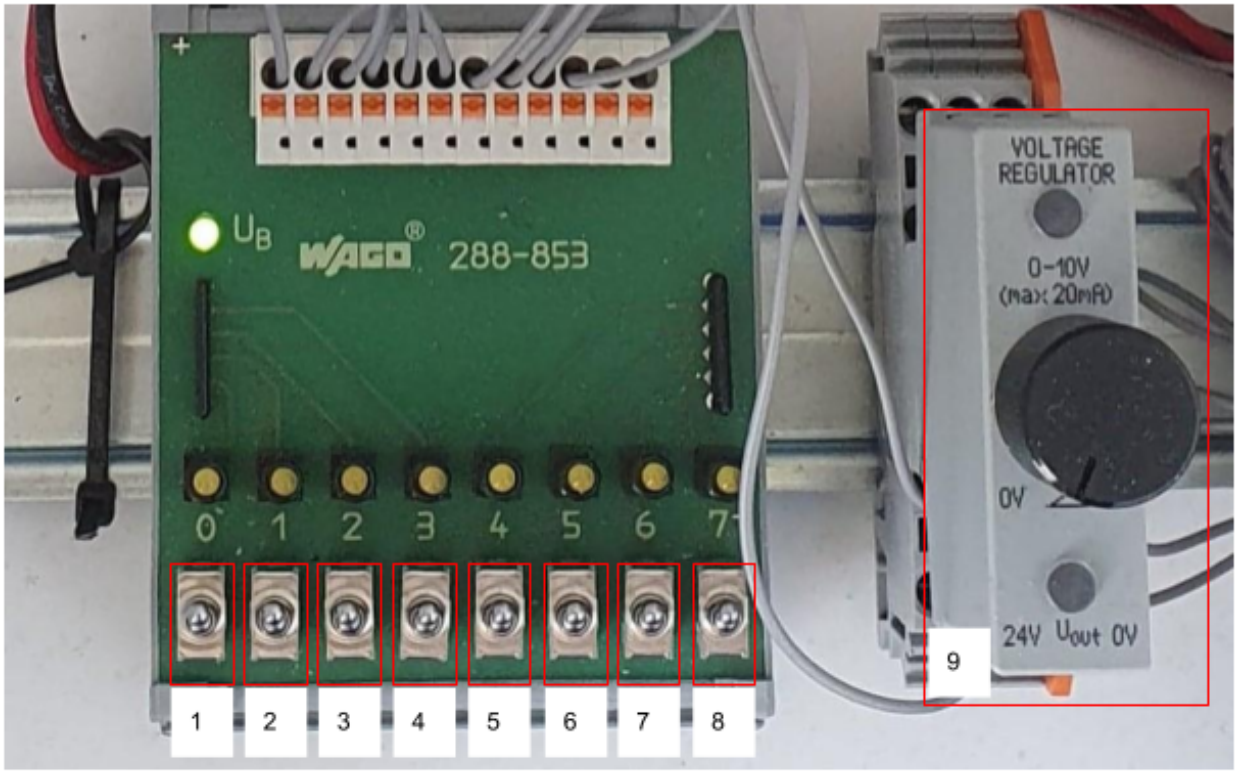

Figure 5.13: Physical Control System for PLC

The following list corresponds to the framed numbers in Figure 5.13, and shows what type of data the PLC receives from the I/O Modules.

1. Binary value 1, or Digital Switch DI1
2. Binary value 2 or Digital Switch DI2
3. Binary value 4 or Digital Switch DI3
4. Binary value 8 or Digital Switch DI4
5. Binary value 16 or Digital Switch DI5
6. Binary value 32 or Digital Switch DI6
7. Digital Switch DI7
8. Digital Switch DI8
9. Analog Regulator 0 - 10V

The control system on the PLC is operated in the following way:

When DI8 is active, the digital switches act as normal and DI1 enables or disables the autopilot. When DI7 is active, and DI8 is not. The digital switches DI1, DI2, DI3, DI4, DI5 and DI6, are using their binary values.

The binary values for controlling the processes are as follows:

- **1**: The regulator controls the effect on the engines AzimuthPort and AzimuthStarboard
- **2**: The regulator controls the temperature on the engines AzimuthPort and AzimuthStarboard
- **4**: The regulator controls the directional angle of the engines AzimuthPort and AzimuthStarboard, given the autopilot is not active,
- **8**: The regulator controls the oil pressure of the engines AzimuthPort and AzimuthStarboard
- **16+1** The regulator controls the effect on the bow thruster engines, given that the autopilot is not active
- **16+2**: The regulator controls the temperature on the bow thruster engines
- **16+4**: The regulator controls the effect on the aft thruster engine
- **16+8**: The regulator controls the temperature on the aft thruster engine
- **32+1**: The regulator controls the level on tank 1 and tank 2
- **32+2**: The regulator controls the level on reserve tank 1 and reserve tank 2
- **32+4**: The regulator controls the temperature on tank 1 and tank 2
- **32+8**: The regulator controls the temperature on reserve tank 1 and reserve tank 2

5.8 Final Setup

The final demo from this project can be seen in Figure 5.14. The website based HMI is transferred on to the TP-600 using Docker and the variables can be simulated using the I/O modules on the PLC.

Figure 5.14: Final result

Chapter 6

Discussion and future work

As most projects, not everything went flawlessly on the first attempt and therefore, we also encountered some obstacles along the way. Through this chapter we discuss some of the problems we faced, possible solutions and future work that could be done towards this project.

6.1 HMI functions

Even though the main functionalities of the HMI are present, there are still a lot of undeveloped functions that would need to be fixed before it could be launched as a finished product. Firstly, for security reasons, the login and authentication would need to be developed to prevent unauthorized access to the HMI.
For larger projects with a combination of larger systems, sections like the Application menu could be fully developed to let the user navigate between these systems' HMIs. The search bar would also need to be implemented, for quick access to different variables and applications.
On the page "Tanks & Pumps", Tank 2-4 are purely cosmetic and have no functionality. To be functional, they would need to be given variables from the PLC.
Lastly, the pages "Settings" and "Help & Support" have no current functionality either, but for future work, it could be implemented as well.

6.2 OpenBridge

The visual and interactive part of the HMI is all made on the basis of the OpenBridge standard. As OpenBridge is a fairly new and an open-source CSS library, currently in beta release, it is not fully complete. The library is meant to be modular and easy to implement but since it is in a relatively early state, there is still a lot of components and styling that have not yet been created for easy implementation. That said, there is a good amount that have been designed visually in Figma (Figma, n.d.), so we are able to create our own components by taking inspiration from the design created there. As this is an open-source design guideline, more components will come over time as more companies use the guideline to make the components they specifically need, as well as the development team will continue to work on it.

By making a lot of components ourselves, we started to notice that the CSS file got quite large and hard to navigate through. From a users perspective you would not need to navigate through or see this file, but for our own development's sake, future development and maintenance, we decided to write the CSS in Sass instead. This is a way of structuring the CSS in a more readable way and easily see what CSS classes that are dependant on each other. We came across the SASS way of structuring by looking through the OpenBridge CSS, as this was the way that they had made it. So to be able to use SASS we had to download the "Live Sass compiler" extension from VsCode.

While using the already made OpenBridge components and classes, they automatically change colors when the theme is changed. The OpenBridge components such as the compass, thrusters, bars etc., are all SVG constructed components and are colored on that basis. For our own custom made pump icon from Figure 4.8, which is a PNG image, we used the CSS filter property to change colors as mentioned. If we were to make a larger project with more custom icons and components, this would not be a substantial or efficient way to color them, as it would require converting all the colors to filters and make new functions for applying them. By using the filters we would also have a hard time getting the exact same color as in the OpenBridge library, although it is possible to get very close. This is because the filters only manipulate the colors that is already in the PNG, similar to photo editing, and does not use the standard RGB values. For future work, it could be beneficial to make the custom images as SVG files and take a closer look into how the OpenBridge components get their coloring and try to adapt this to our custom icons.

6.3 Trying a new framework

As this was going to be a single page HMI, which means that everything that is visually displayed is rendered from only one HTML file, we got recommended to use Vue.js by a full stack web developer. The reason for this is that while having the website written in the Vue.js framework, the website would remove all content that is not currently displayed from the HTML file, while a regular HTML file would just "hide" it. This would greatly reduce the process power needed to render the website. However after working with Vue.js for a while, we realised it would not work with the socket communication we had already made. This was due to the removing of the non-displayed variables, as a system like this would need to updated all variables continuously for the processes to function in relation to each other, and give notification of possible system fails. Therefore we decided to move away from Vue.js and return back to the original plan of a more basic website development style.

6.4 Graphing Library

In preparation to make the depth graph, quite a lot of research was done to find a solution with the current knowledge we had towards JavaScript. Unfortunately we realized that it would be far too complex to make, and time consuming to learn. Therefore we did some more research for JavaScript libraries that would aid us in plotting our graph. There were a few good available libraries, but most of them had limitations on real time plotting, except for the Plotly library, which we ended up using.

6.5 PLC program

Since we had little prior experience with PLC programming, and never seen WAGO's own software, we found it necessary to do some training with it before starting the project. Due to the ongoing Covid-19 pandemic, we were unable to attend any physical meetings with WAGO. As a solution we managed to participate on a three day e!COCKPIT course via Microsoft Teams, hosted from the training center in Germany. During this course, we expanded our knowledge towards PLC programming, and it gave us a good understanding on how to start the project.

When we started creating processes in the PLC, we created processes with redundancy and safety measures like: emergency stop, timers and individual starting cycles, so that certain systems was unable to start before their requirements were met. One example of this, was that the engines could only run if the temperature or the oil pressure was within the limits, and the emergency stop was false. However since the PLC program only simulates processes for the HMI, it became an obstacle having such complex system without actual industrial hardware. Especially since we were only able control one function at a time with our equipment. Additionally, it was not an objective of this project (Project Aims and Objectives 1.2). As a result we chose not to focus on redundancy or safety regarding the processes. Instead we focused on making values increase or decrease depending on the input from the user, as it would show the potential of the HMI better. For an actual real life scenario, the PLC program would require much more work, planning and safety measures.

6.6 MTP

The general idea of the MTP standardization is in our opinion very good. It shows promising qualities, regarding the future of efficient engineering. It seems that MTP would benefit the manufacturer and the installer, as well as the user. However, the standard is not yet fully finished, and that includes the unfinished MTP library. Although many manufacturers similar to WAGO, such as Siemens, Festo, Schneider and ABB also supports this standard, there is very little information regarding the use of MTP. This made it difficult to understand how to properly use the standard, and a lot of time went into learning about it.

One of the key elements of the MTP, is how flawless modules can be visualized in a MTP supported control-/visualization system. Unfortunately HTML based HMI, like our own, does not fully support the innovative functions of MTP visualization, making the HMI unable to fully utilize the functions MTP provides. That said, we managed to get the communication between the MTP elements and the HMI, which was the actual object for this project. As MTP continues to be developed, this may certainly be one of the topics to advance.

6.7 Choosing a OPC-UA client

There were several different ways we could go when choosing a client framework. We took a look into **Python** (The Python Software Foundation, n.d.), an easy to use programming language that we had some experience with, in addition to having some nifty OPC-UA tutorials online.

Unfortunately, Python seemed to lack integration with web page communication, as making two programming languages talk to each other (JavaScript, Python), did not seem like the most efficient route and the idea was quickly dismissed.

Another potential route, was a client built on **Node-RED** (OpenJS Foundation, n.d.-e), which is a low-code programming tool used to connect API, online-services, devices and other hardware together in a visual styled way. Node-RED was also recommended by WAGO as our supervisor had some experience with it. We used a lot of hours getting to know Node-RED, as it seemed like it had everything we needed: OPC-UA add-ons, HTML compatible, and was a plug and play type programming language. The progress nevertheless came to a halt when we discovered that using external CSS and JavaScript for a HTML web page, was not possible, at least not to our knowledge. We needed a different approach.
That is when we decided on **node-opcua**, as it was based on NodeJS and would fit great into our project (see OPC-UA 3.5.3).

6.8 Node OPC-UA client

Some improvement to the OPC-UA client would possibly need to be made in the future, as this is a 1 subscription manual client with around 20 variables. It is more common to use only a few variables per subscription, and it is not possible to use 1 subscription per lets say 100 variables, as that would most likely result in crashes. On a industrial plant, there are up to thousands of variables that needs to be monitored and controlled, meaning there needs to be more subscriptions in the client. There have also been developed some functions that can extract entire directories with variables, removing the need to write the address of each variable separately (Rossignon, n.d.-b). Making the OPC-UA client semi- or fully automated would be the next step of integrating it into industrial automation.

6.9 TP 600

The TP-600 (Touch Panel 600 3.3.3), which is the touch panel we used in this project, is mainly used for visualization through the e!COCKPIT program, and unfortunately was not the best for handling websites. This was made obvious after we ran a prototype of the project on the touch panel and the processor capacity was at max at all times. This can be seen in Figure B.1 where the CPU is running at 122%. We found that the touch panel did not handle constant, smooth or big scale visual changes. This was most likely due to a slow processor with few cores, which was not made to handle a website of this scale. We also tried switching web browsers from the pre-installed QT web browser to a Chromium one, but only got worse results. We got better results by toning down on the visual changes, as seen in Figure B.2, but this issue could probably get a more permanent fix by using one of WAGO's newer models of touch panels, which have more powerful and faster processors (Wago, n.d.-c).

Chapter 7

Conclusion

This thesis aimed to present a functioning demo HMI based on the OpenBridge design system, that simulates maritime processes within the MTP framework on the PFC 200, while using the OPC-UA communication protocol.
After a combined total of over 1600 hours of learning, testing and troubleshooting, in addition to over 3500 lines of code in HTML, CSS, JavaScript and Structed-Text, we can conclude that this project has been a success.
The results of this project has been approved by our supervisor from WAGO, Tor Erik Næbb with great appreciation, and shown the possibility for quality regarding HMIs. As for Module Type Packaging, we have successfully managed to implement the MTP function-blocks into our PLC program, transferring their variables like alarms and analog values from the PLC to the HMI and back. That being said, we have not used MTP completely as intended, as MTP is still a work in progress in the automation industry and one of its main functionalities is to be able to export the visualization from WAGO's e!COCKPIT and use it together with other manufacturers visualizations.
As our objective was to use OpenBridge for visualization, we could not rely on the e!COCKPIT visualization program, therefore not testing the full potential of MTP. Nevertheless we manged to use some of its functionalities in our project, and seeing as WAGO's main priority was to have a visually appealing and functioning HMI, we believe we have delivered a project above expectations.

Appendix A

Full Project on GitHub

As it would not be favourable to put 3500 lines of code in the thesis, we have made the whole project available on GitHub. Every necessary part of this project can be viewed and accessed through the link or QR code below.

https://github.com/Wago-Bachelor/WebVisu

GitHub QR code

Appendix B

TP600 Processor Performance

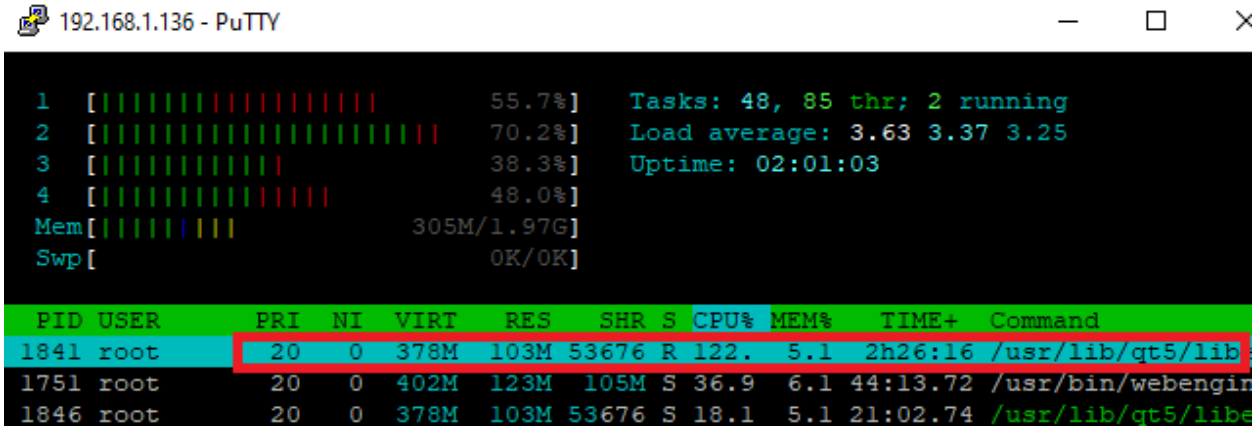

Figure B.1: Wago TP 600 CPU performance at 122%

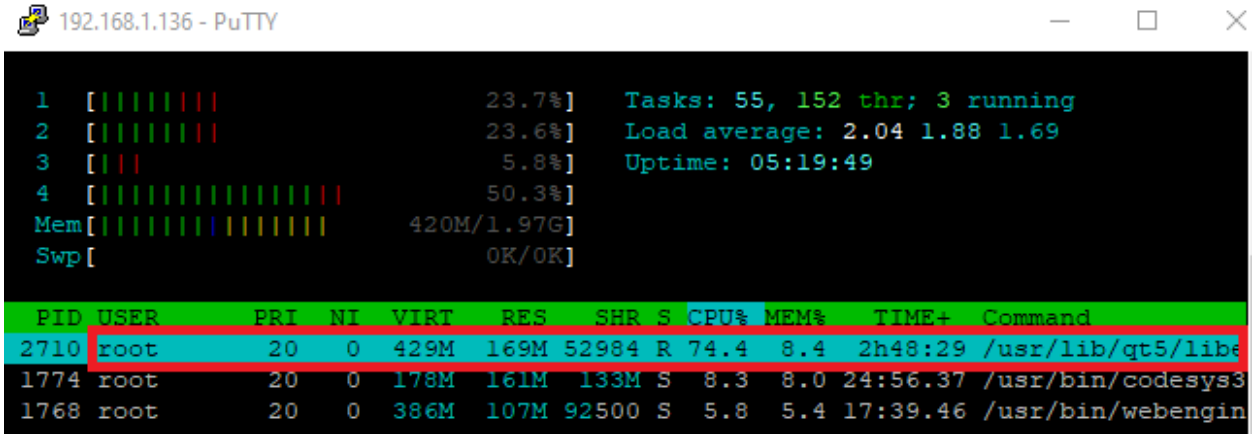

Figure B.2: Wago TP 600 CPU performance at 74%

Appendix C

Flow charts

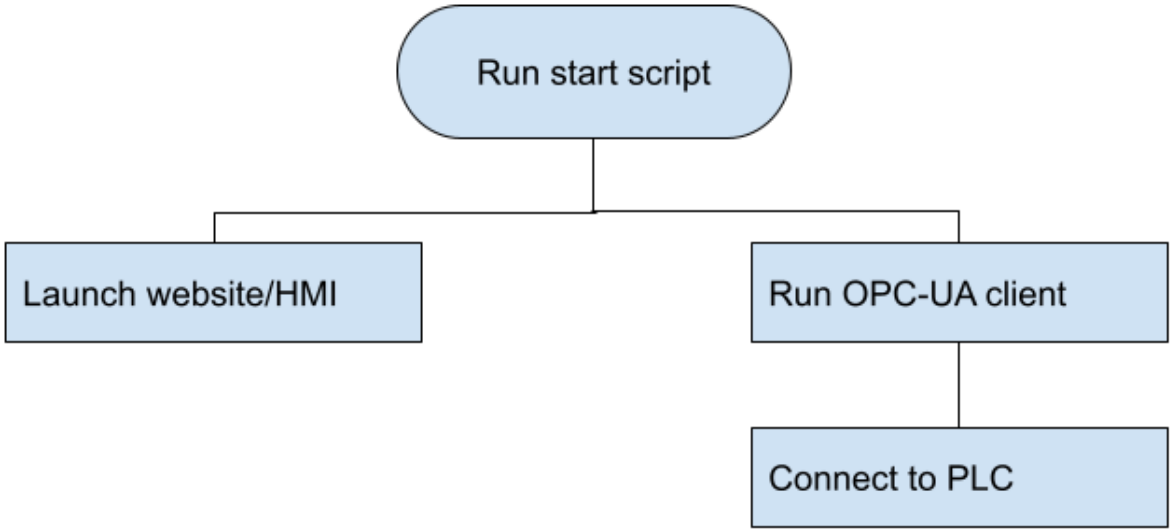

Figure C.1: Start sequence

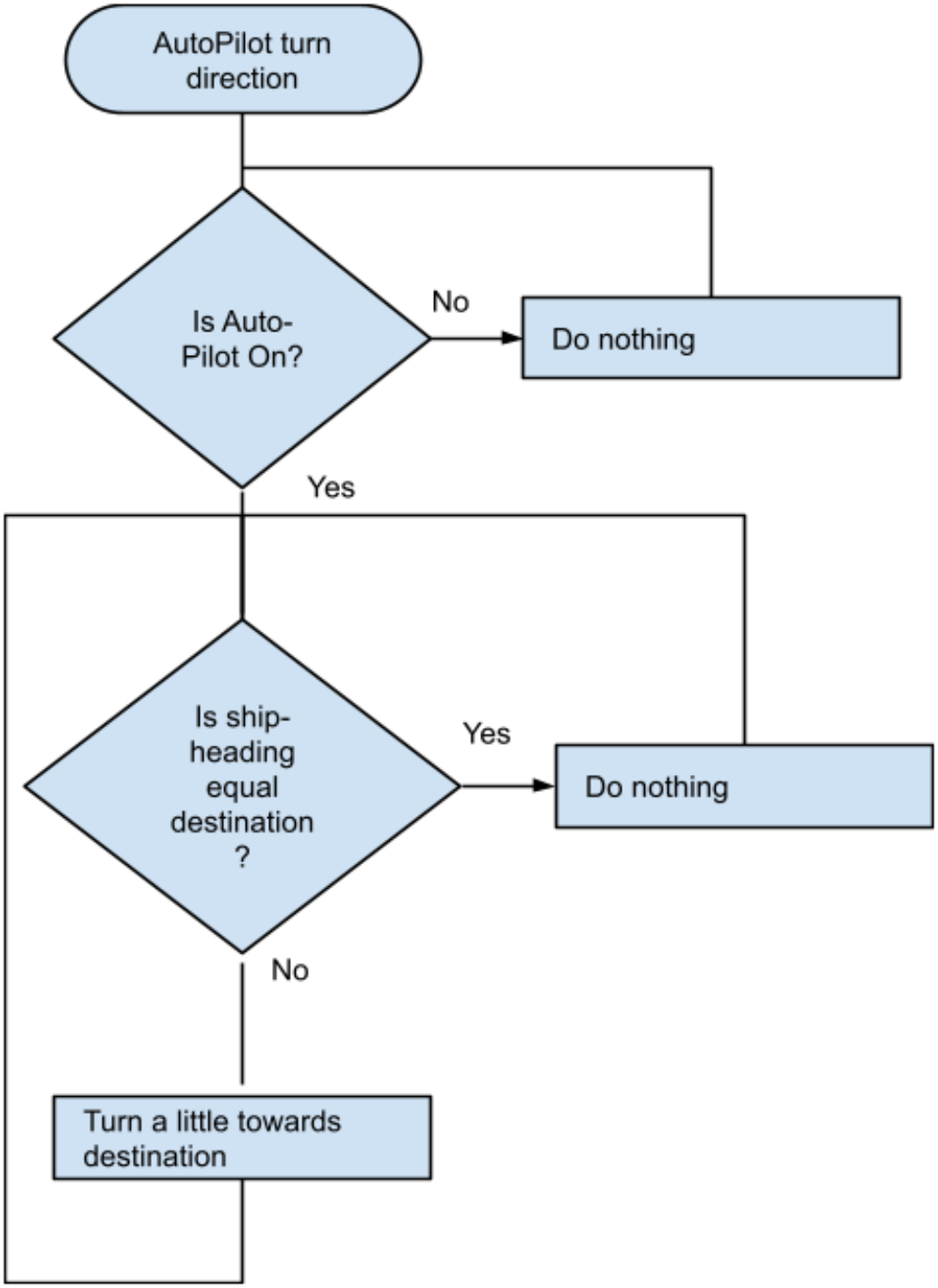

Figure C.2: Auto Pilot flow chart

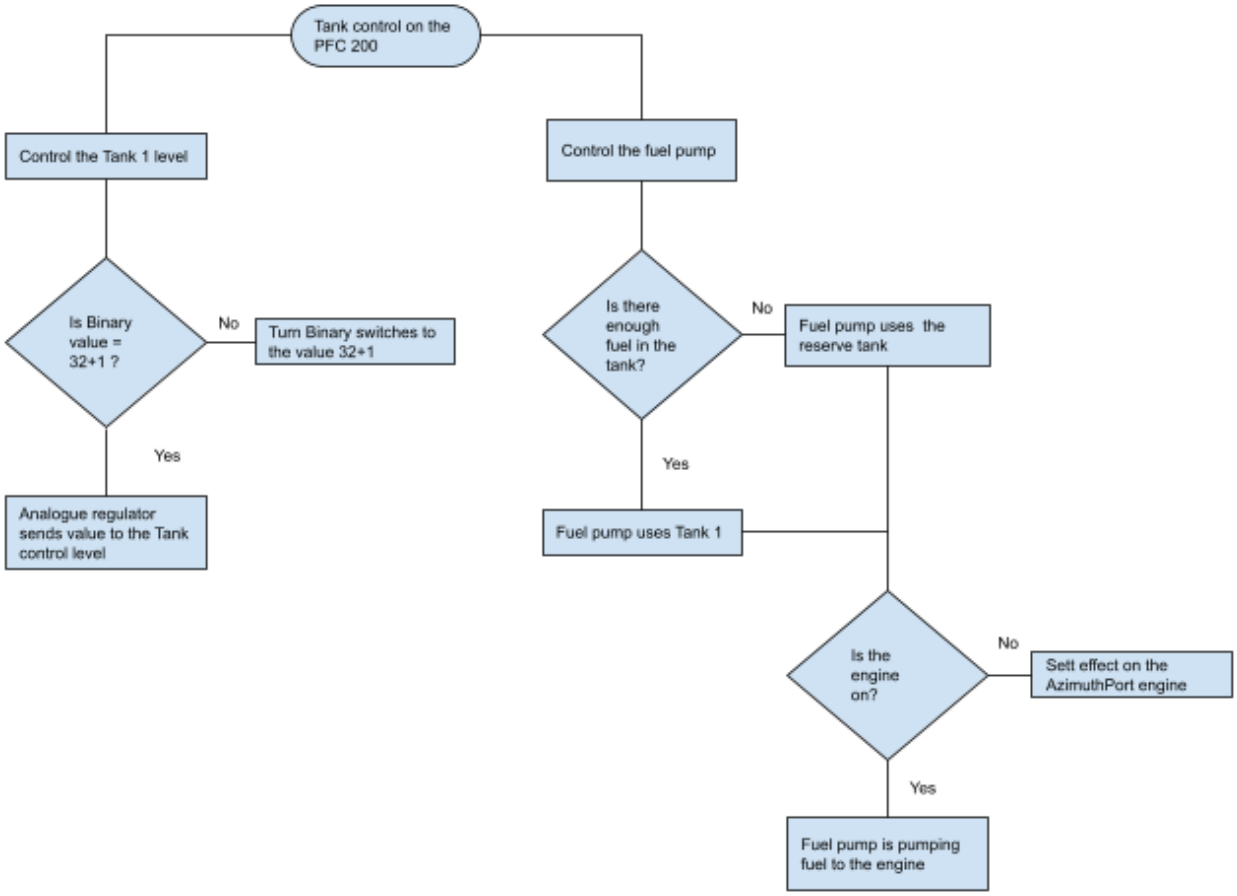

Figure C.3: Tank control flow chart

Appendix D

Communication between parts

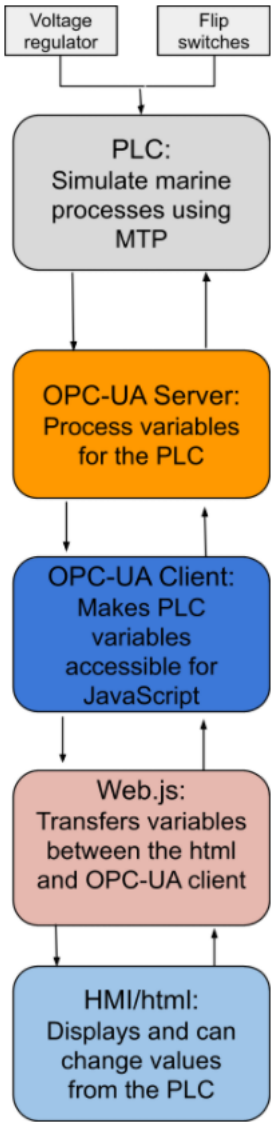

Figure D.1: Communication channels

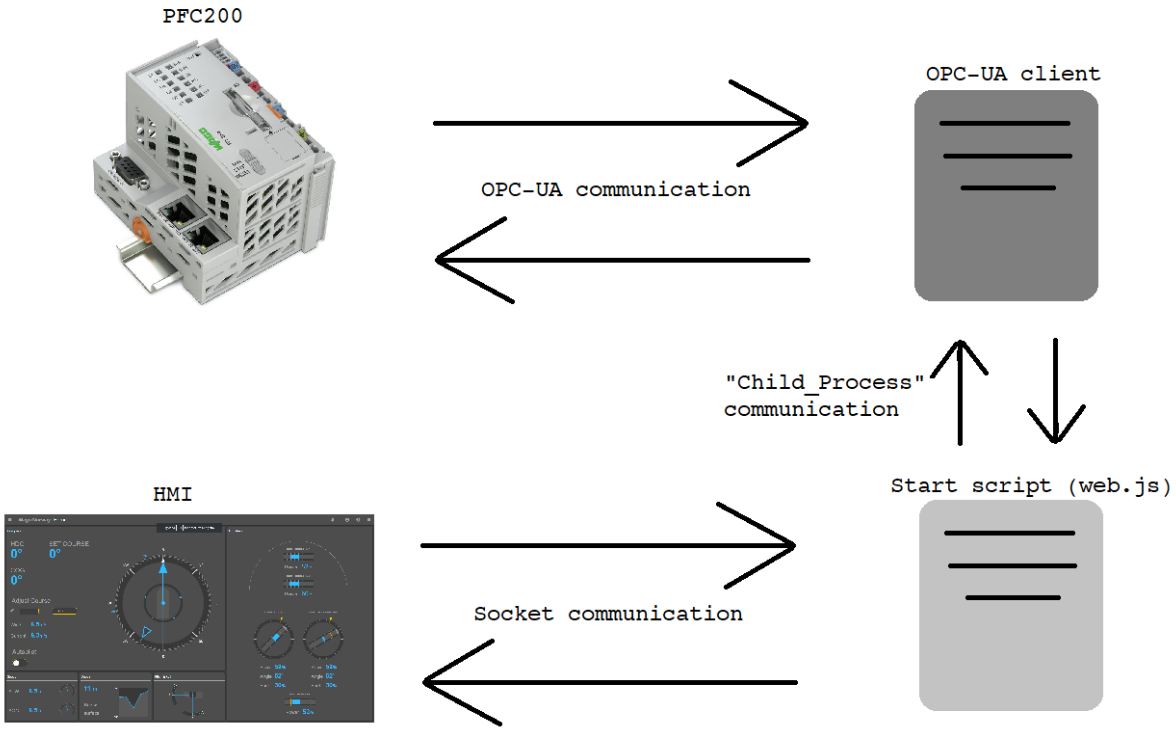

Figure D.2: Communication model

Appendix E

Component list

Item	Type	Part Number	Amount	Manufacturer	Usage
Power cables	230V (2m)	67078	2	Kjell & Company	Connect Power Supply
Ethernet cables	Cat 6 (2m)	756-1250	2	WAGO	Communication
I/O module	AI module	750-459	1	WAGO	Process signals
I/O module	AO module	750-559	1	WAGO	Process signals
I/O module	End module	750-600	1	WAGO	Completes circuit
I/O module	Digital module	750-1506	1	WAGO	Process signals
Connection cable	60V (5m)	39515	1	Kjell & Company	Connect components
Mounting plate	Mounting system	(N.A.)	1	WAGO	Mounting hardware
Power Supply	24V	787-2850	2	WAGO	Power supply
PLC	PFC 200	750-8212	1	WAGO	Programming
Switch Interface	8-CH. Switch	288-853	1	WAGO	Toggle switches
Touch Panel	TP600	762-5205	1	WAGO	Display
Voltage regulator	24V regulator	286-914	1	WAGO	Regulates 0-10V

Bibliography

Behance. (n.d.). *Putty logo*. Retrieved May 22, 2021, from https://www.behance.net/gallery/70251739/PuTTY-svg-logo

Bø, T. I. (n.d.-a). *Openbridge css*. Retrieved May 20, 2021, from https://gitlab.com/openbridge/openbridge-css

Bø, T. I. (n.d.-b). *Openbridge web components*. Retrieved May 20, 2021, from https://gitlab.com/openbridge/openbridge-web-components

Braun, K. (2016, October 13). *Wago plc's onboard opc ua server tutorial with ignition scada*. Youtube. Retrieved May 22, 2021, from https://www.youtube.com/watch?v=wSpddb_XCWQ

Buna, S. (2017, June 8). *Node.js child processes: Everything you need to know*. Retrieved May 22, 2021, from https://www.freecodecamp.org/news/node-js-child-processes-everything-you-need-to-know-e69498fe970a/

Dan's Tools. (n.d.). Rgb to hex color coverter. Retrieved April 17, 2021, from https://www.rgbtohex.net/

Docker. (n.d.). *Docker*. Retrieved May 22, 2021, from https://www.docker.com/resources/what-container

Engebretsen, B. (2017). *Automatiseringsanlegg del iii, pls systemer*. Cybernetics Forlag.

ExpressJS. (n.d.). *Express - node.js web application framework*. Retrieved May 22, 2021, from https://expressjs.com/

Fachbereich Industrielle Informationstechnik. (n.d.). *Automation engineering of modular systems in the process industry*. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik. Retrieved March 4, 2021, from https://www.vdi.de/

Figma. (n.d.). *Figma*. Retrieved May 20, 2021, from https://www.figma.com/

Hack Reactor. (2018, October 18). *What is javascript used for?* Retrieved May 22, 2021, from https://www.hackreactor.com/blog/what-is-javascript-used-for

Inductive Automation. (n.d.). *Ignition*. Retrieved May 22, 2021, from https://inductiveautomation.com/ignition/

Inductive Automation. (2018, August 10). *What is hmi?* Retrieved May 22, 2021, from https://www.inductiveautomation.com/resources/article/what-is-hmi

Inductive Automation. (2020, February 24). *What is plc?* Retrieved May 22, 2021, from https://www.inductiveautomation.com/resources/article/what-is-a-PLC

MDN Web Docs. (n.d.). *How does css actually work?* Retrieved May 22, 2021, from https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works

Mozilla. (2021, May 15). *Mozilla developer networks*. Retrieved May 22, 2021, from https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5

NodeJS Docker Team. (n.d.). *Arm32v7/node*. Retrieved April 24, 2021, from https://hub.docker.com/r/arm32v7/node/

Novotek. (n.d.). *Opc and opc ua explained*. Retrieved May 22, 2021, from https://www.novotek.com/uk/solutions/kepware-communication-platform/opc-and-opc-ua-explained/

OPCFoundation. (n.d.). *Unified architecture*. Retrieved May 22, 2021, from https://opcfoundation.org/about/opc-technologies/opc-ua/

OpenJS Foundation. (n.d.-a). *About nodejs*. Retrieved May 22, 2021, from https://nodejs.org/en/about/

OpenJS Foundation. (n.d.-b). *Child_process*. Retrieved May 22, 2021, from https://nodejs.org/api/child_process.html

OpenJS Foundation. (n.d.-c). *Dockerizing a node.js web app*. Retrieved May 22, 2021, from https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

OpenJS Foundation. (n.d.-d). *Http*. Retrieved May 22, 2021, from https://nodejs.org/api/http.html

OpenJS Foundation. (n.d.-e). *Node-red*. Retrieved May 22, 2021, from https://nodered.org/about/

Oslo School of Architecture and Design. (n.d.). *Openbridge design system*. Retrieved May 10, 2021, from http://www.openbridge.no/

Pelling, S. (n.d.). *The net ninja*. YouTube channel. Retrieved May 10, 2021, from https://www.youtube.com/channel/UCW5YeuERMmlnqo4oq8vwUpg

Plotly. (n.d.). *Plotly - graphing libraries*. Retrieved May 14, 2021, from https://plotly.com/graphing-libraries/

PuTTY. (n.d.). *Putty*. Retrieved May 22, 2021, from https://www.putty.org/

Ramanathan, R. (n.d.). *The iec 61131-3 programming languages features for industrial control systems*. Retrieved May 4, 2021, from https://ieeexplore.ieee.org/document/6936062

Roberts, B. (n.d.). *Thenewboston*. Retrieved May 10, 2021, from https://www.youtube.com/channel/UCJbPGzawDH1njbqV-D5HqKw/featured

Rossignon, E. (n.d.-a). *Datatype*. Retrieved May 22, 2021, from https://node-opcua.github.io/api_doc/2.0.0/enums/datatype.html

Rossignon, E. (n.d.-b). *Node-opcua*. Retrieved May 22, 2021, from https://node-opcua.github.io/

Smith, J. L., & Granell, C. (n.d.). *Best code editors 2020: Your guide to the top options*. Retrieved May 22, 2021, from https://www.creativebloq.com/advice/best-code-editors

Socket.IO. (n.d.). *Socket.io*. Retrieved May 22, 2021, from https://socket.io/

Sonntag, B. (n.d.). *Codepen home css filter generator to convert from black to target hex color*. Retrieved April 17, 2021, from https://codepen.io/sosuke/pen/Pjoqqp

Stackify. (n.d.). *Learn javascript*. Retrieved May 24, 2021, from https://stackify.com/learn-javascript-tutorials/

The Python Software Foundation. (n.d.). *Python*. Retrieved May 22, 2021, from https://www.python.org/

The Safety. (n.d.). *Javascript logo*. Retrieved May 22, 2021, from https://thesafety.us/javascript-disable-firefox-chrome-safari-opera

Visual Studio. (n.d.-a). *Live sass compiler*. Retrieved May 22, 2021, from https://marketplace.visualstudio.com/items?itemName=ritwickdey.live-sass

Visual Studio. (n.d.-b). *Live server*. Retrieved May 22, 2021, from https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

Visual Studio. (n.d.-c). *Visual studio code*. Retrieved May 22, 2021, from https://code.visualstudio.com/

Visual Studio Code. (n.d.-a). *Extension marketplace*. Retrieved May 24, 2021, from https://code.visualstudio.com/docs/editor/extension-marketplace

Visual Studio Code. (n.d.-b). *Visual studio code*. Retrieved May 22, 2021, from https://code.visualstudio.com/

w3. (n.d.-a). *Html & css*. Retrieved May 22, 2021, from https://www.w3.org/standards/webdesign/htmlcss

w3. (n.d.-b). *Html5*. Retrieved May 22, 2021, from https://www.w3.org/html/logo/index.html

W3schools. (n.d.). *W3schools online web tutorials*. Retrieved May 10, 2021, from https://www.w3schools.com/

Wago. (n.d.-a). *Ahead of the curve with mtp*. Retrieved May 10, 2021, from https://www.wago.com/global/digitization/convertibility

Wago. (n.d.-b). *Wago logo*. Retrieved May 3, 2021, from https://www.wago.com/

Wago. (n.d.-c). *Wago products*. Retrieved May 3, 2021, from https://www.wago.com/global/products

Wago. (n.d.-d). *Wago software*. Retrieved May 3, 2021, from https://www.wago.com/global/c/software

WAGO Norge AS. (n.d.). *Wago norge - github*. Retrieved April 24, 2021, from https://github.com/Wago-Norge