Master Thesis

CTU
CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3 Faculty of Electrical Engineering
Department of Measurements

Hydronic Heating Testbed Automation

Bc. Jiří Cvrček

Supervisor: Ing. Jiří Dostál
Field of study: Computer Engineering
May 2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Cvrček**	Jméno: **Jiří**	Osobní číslo: **420794**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra měření**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Počítačové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Automatizace pro testovací zařízení komponent otopných systémů

Název diplomové práce anglicky:

Hydronic Heating Testbed Automation

Pokyny pro vypracování:

1. Study (building) automations systems architectures. Review and enhance given hydronic testbed proposal.
2. Design an automation system for a defined hydronic heating testbed using a programmable logic computer (PLC). The PLC should be able to run a separate process for a custom-made predictive controller.
3. Develop communication stack for all provided peripherals. The automation system software should be developed flexible in terms of changing periphery set, safety critical tasks and override policies.
4. Prepare SCADA operated by an enclosed graphic touch display.

Seznam doporučené literatury:

[1] Domingues, P. et al.: Building automation systems: Concepts and technology review – Computer Standards & Interfaces, 2016.
[2] Samad, T. et al.: System architecture for process automation: Review and trends – Journal of Process Control, 2007.
[3] Zelenka, D.: Cyber-physical One-pipe Hydronic Heating Testbed – CTU, 2019.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jiří Dostál, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.01.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce:
do konce letního semestru 2019/2020

Ing. Jiří Dostál podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.
Datum převzetí zadání   Podpis studenta

Acknowledgements

I would like to thank everyone who helped me with their ideas during this work. Above all, I want to thank my supervisor, Ing. Jiří Dostál. Next big thanks goes to my colleague Tomáš and Ondra, who constructed the testbed and helped with the schematics and photos.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 24. May 2019

Abstract

This work is about automating the control of a heating system in which active components need to be controlled. At first I was introduced to the hydraulic testbed connection and then started to select electrical components. There was a need to select sensors to measure system pressure, temperature, and water flow, followed by temperature and air flow measurements in the testbed air path. In addition to the individual components, I chose a controller that handles all components and provides a communication interface to the application in Matlab Simulink.

For operation and communication with peripherals I created a program running on the control unit. The main feature was easy configuration of all peripherals and simple API for communicating with the client program. The controller application is programmed in Python and communicates via the ethnernet network with the simulation client in Matlab Simulink.

Keywords: heating system, one-pipe heating, control unit, communication stack

Supervisor: Ing. Jiří Dostál

Abstrakt

Tato práce se zabývá automatizací řízení otopného systému, v kterém je nutné řídit aktivní komponenty. Nejprve jsem se seznámil s hydraulickým zapojením testbedu a následně začal vybírat elektrické komponenty. Bylo potřeba vybrat senzory pro měření tlaku, teploty a průtoku vody v systému, následně i měření teploty a průtoku vzduchu ve vzduchové cestě testbedu. Kromě jednotlivých komponent jsem vybíral řídicí jednotku, která obsluhuje všechny komponenty a poskytuje komunikační rozhraní pro komunikaci s aplikací v Matlab Simulinku.

Pro obsluhu a komunikaci s periferiemi jsem vytvořil program běžící na řídicí jednotce. Hlavním poaždavkem byla jednoduchá konfigurace všech periferií a jednoduché API pro komunikaci s klientským programem. Aplikace na straně řídicí jednotky je naprogramována v Pythonu a komunikuje prostřednictvím sítě ethnernet se simulačním klientem v Matlab Simulinku.

Klíčová slova: otopný systém, jednopotrubní vytápění, řídicí jednotka, komunikační kanál

Překlad názvu: Automatizace pro testovací zařízení komponent otopných systémů

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Heating

In nowadays, a warm place to stay is one of the basic comfort needs of people. People use mostly central heating system for heat up air in the room. The principle is to heat up water in a boiler and transport it to a heat exchanger in desirable place. The water heating system and heat exchanger have been continuously developing, but the transport network topology stays almost the same. There exist two systems, one-pipe and two-pipe. Both of them have their positives and negatives.

1.1.1 Two-pipe Hydronic Network

Most common type of heat distribution system is a two-pipe system. Boiler and all heat exchangers are parallel connected, thus the input temperature of the water is the same in each heat exchanger. Cold water returns directly to the boiler. Pressure in a pipe is generated by a pump in the circuit. This type of network is used in industry and is used by designers and plumbers. This system is widely used and advanced technology. [1]

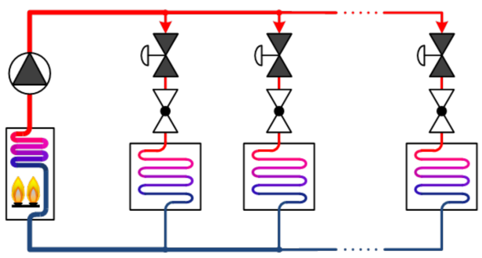


Figure 1.1: Two-pipe passive heating network. [2]

1.1.2 One-pipe Hydronic Network

Nowadays, Czech designers are not interested in one-pipe heating systems. They prefer two-pipe heating system, as we know from most local buildings. This heating system is suitable for office buildings.

Hot water preparation is the same as for a two-pipe heating system. The main difference is in the distribution network. In one-pipe heatign system hot water is prepared in a boiler and circulated in a primary circuit. Heat exchangers are connect in series in the primary circuit. Heat exchangers are connected to a primary circuit with a double T fitting, one inlet to the secondary circuit and one outlet from the secondary back to the primary pipe. There is a pump in the secondary circuit that sucks in a hot water from primary circuit. The hot water is cooled in the exchanger and the now cold water is returned back to the primary circuit. [1]

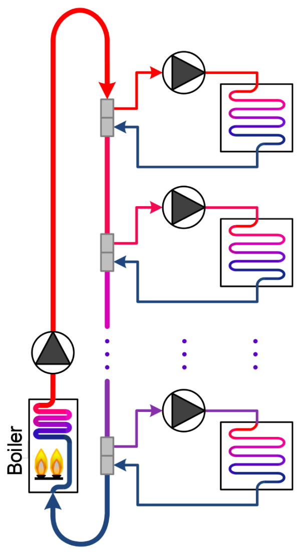


Figure 1.2: One-pipe active heating network. [2]

1.2 Research of existing solutions

There are protocols for communication between client software and hardware. One of the protocols is OPC (OLE for Process Control) [3]. OPC is a united protocol used to exchange data between hardware and client software. The hardware part consists of OPC server and hardware. The server communicates with the hardware via a hardware communication protocol, such as Modbus, and provides the data to the client via OPC protocol. This method eliminates problems with different communication protocols. The disadvantage is that each component has its OPC server. Multiple components can share one OPC server. Configuring a shared OPC server is more complex. The programmer must know the hardware communication protocol and the OPC protocol. OPC is the only communication protocol, it cannot control the components offline, for example, in case of water overheating, turn off the boiler without the client command.

Another existing solution for controlling hardware components is Mervis. Hardware component behavior can be programmed using a graphical interface with function blocks. It is not possible to connect with the client software and control the individual hardware components manually. Data logging can be stored in the cloud Mervis database, which is for a fee based on the number of data points.

Subject of this thesis is to create a system for automation of heating systems. The system allows easy hardware component configuration and communicate over Ethernet with client. The system monitors the selected critical values and, if exceeded, performs the steps to a safe state.

Chapter 2

Hardware

2.1 Testbed

A testbed is a platform for conducting transparent and replicable testing of scientific theories. The testbed is designed for heating system testing. The testbed was primarily designed to test the control of one-pipe heating systems, but it can also be used to test two-pipe heating systems. There are several active components that are controlled according to the values measured by the passive components. The testbed has two parts, electric and hydraulic. I worked on the design and construction of electrical parts.

To design the electrical part of the testbed, I chose and ordered some components. The components had to be selected so they would meet both hydraulic and electrical criteria, such as pipe connection threads or suitable supply voltage. I participated with my colleagues in the design of the electrical wiring so that all the components could be connected. Peripheral power is switched on by relay. This has the advantage that the testbed is always on and if we want to make a measurement, we turn on the relay using the UniPi output. If you need to reset the peripherals, just turn the power off and on remotely.

The testbed that was created during the work is shown in Fig 2.1. The electric part is shown in Fig 2.2 and the hydraulic part is shown in Fig 2.3

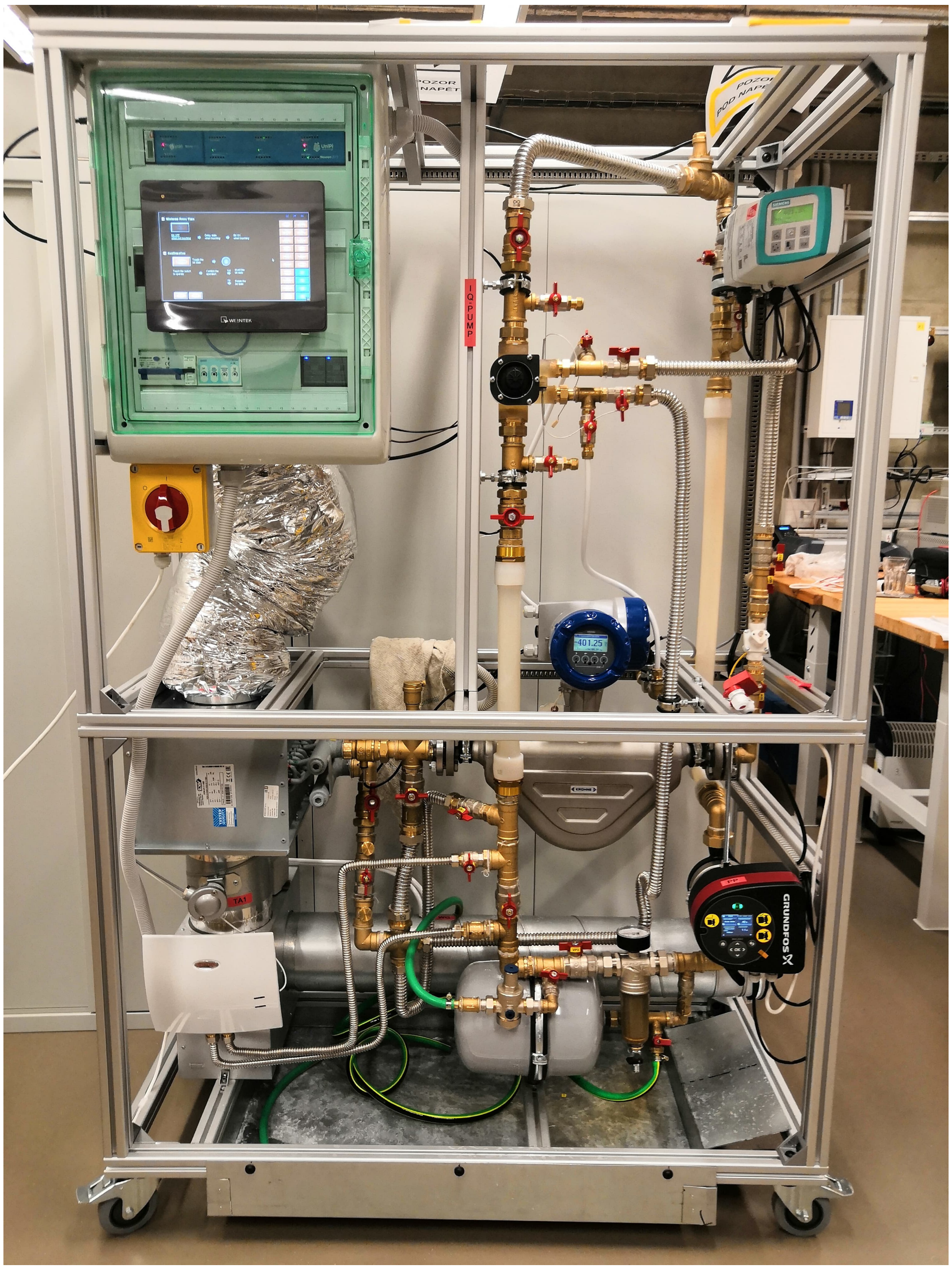


Figure 2.1: Testbed

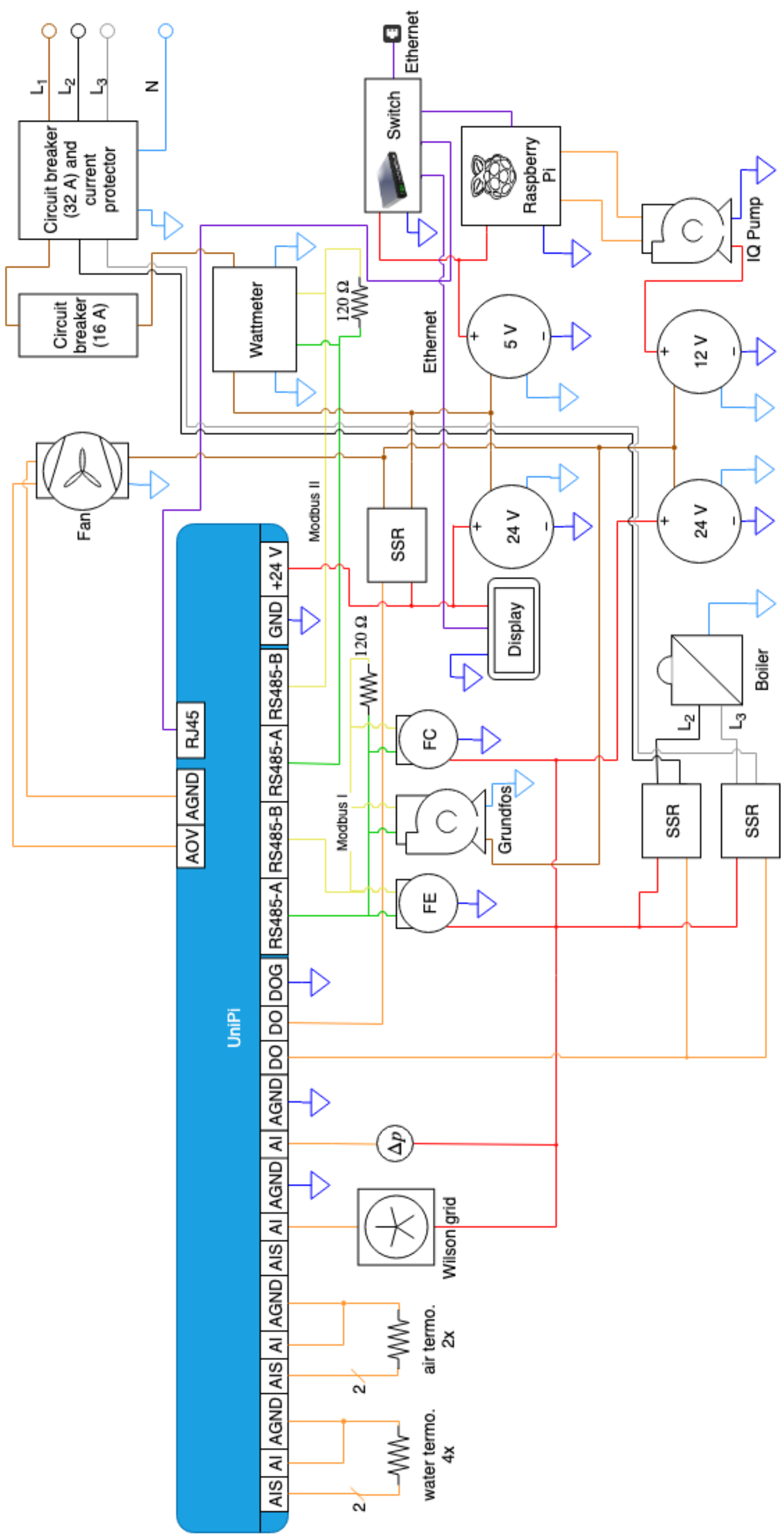


Figure 2.2: Electric testbed scheme

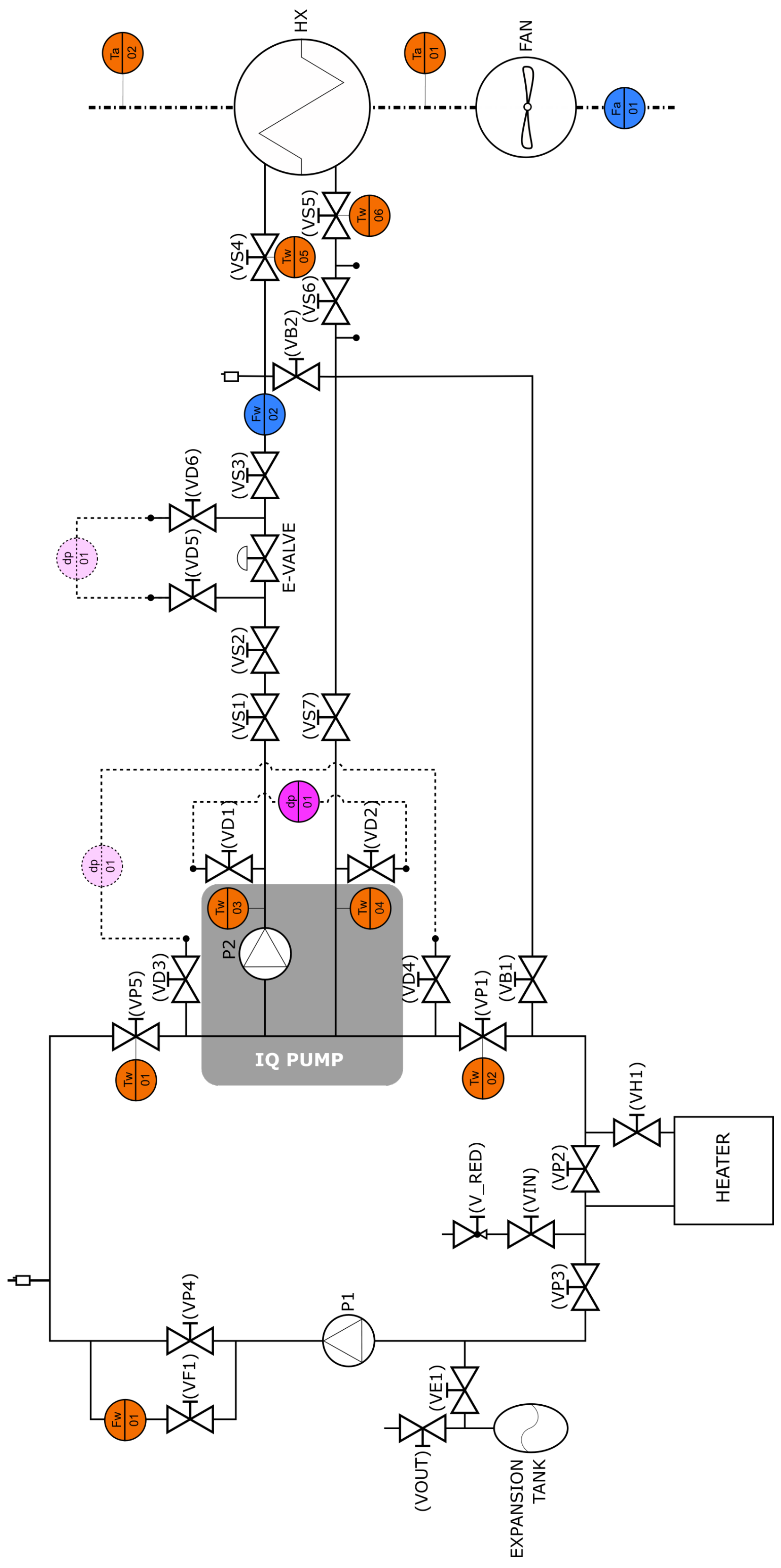


Figure 2.3: Hydraulic testbed scheme

2.2 Components

At the begin of work, I must chose an electric components for the testbed. Many components are passive, such as thermometers, flowmeters or pressure sensor. Active components include water pump, relays and fan.

2.2.1 Thermometers

Many components are thermometers with resistance sensor PT1000. In this project, use a four-wire connection.

Thermometers from SENSIT s.r.o. are used to measure water temperature in Fig. 2.4. These thermometers are screwed into the measuring tubes in the valves. The temperature range is chosen according to the water temperature with a sufficient reserve, from 0 °C to 105 °C.

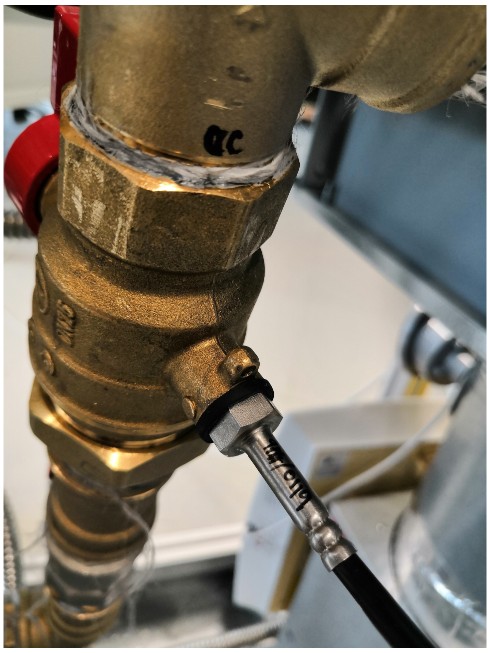


Figure 2.4: Water thermometer

The temperature of air is measured by thermometers from RAWET s.r.o. in Fig. 2.5, these thermometers are from custom production. Thermometers contain a PT1000 sensor, whose temperature range is up to 85 °C. For these temperature sensors is required a fast response to changes ambient air temperature.

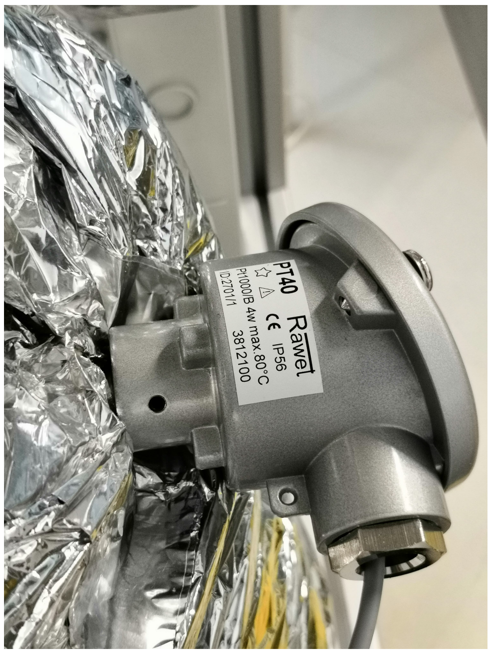


Figure 2.5: Air thermometer

2.2.2 Pressure sensors

In the testbed, there is a pressure sensor for measuring absolute and differential fluid pressure. These values are used to determine the amount of energy transferred.

For measuring differential pressure we use MEAS D5154-000005-005PD in Fig. 2.6. This sensor has the maximum allowable differential pressure 5 PSI. To connect this sensor we use a current loop with range 4-20mA.

The absolute pressure sensor is not used at this time. An analog indicator is used to check the absolute pressure.

2.2.3 Flowmeters

In this project, we need to know the amount of water flowing for proper control.

For measuring water flow in the primary circuit we use inductive flowmeter

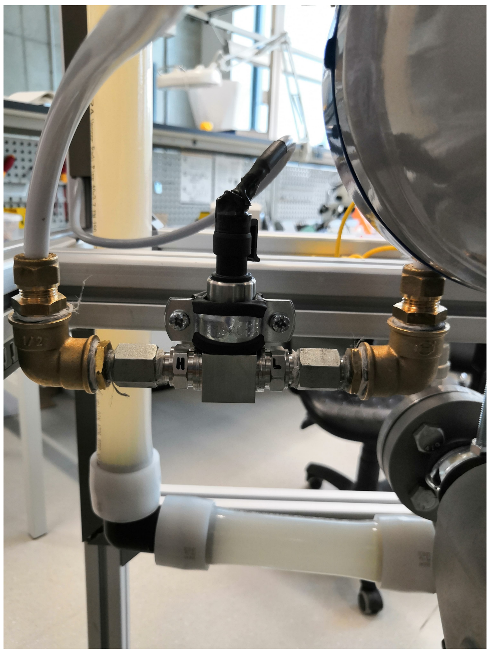


Figure 2.6: Differential fluid pressure.

SITRANS F M MAG 1100 from Siemens in Fig. 2.7. To connect this flowmeter we use RS485 with Modbus protocol. In the flowmeter documentation, it is possible to find a list of all registers for configuration and reading of measured values.

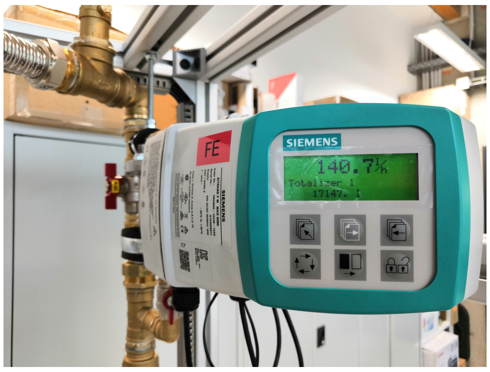


Figure 2.7: SITRANS F M MAG 1100

For measuring water flow in the secondary circuit we use Coriolis mass flowmeter Optimass 6400 S15 from Krohne in Fig. 2.8. To connect this mass flowmeter we use RS485 with Modbus protocol. In the mass flowmeter documentation, it is possible to find a list of all registers for configuration and reading of measured values.

To measure the heat energy transmitted by the water-air exchanger, it is necessary to know the amount of air flow and temperature change.

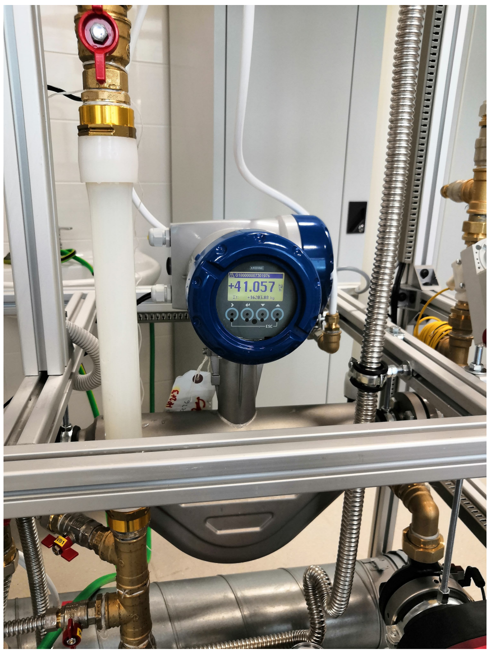


Figure 2.8: Optimass 6400 S15

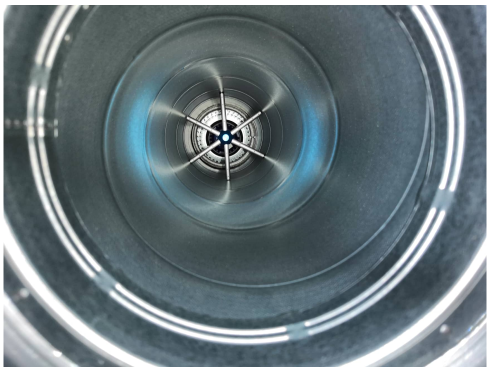


Figure 2.9: AL-grid from AIRFLOW

The air velocity is measured by the AL-grid from AIRFLOW in Fig. 2.9. The AL-grid evaluates the differential pressure in the air duct, which is converted using pressure transducer PTSXR in Fig. 2.10. To connect pressure transducer we use a current loop with range 4-20mA. The pressure transducer is powered by 24V DC.

2.2.4 Frequency converter

The air flowing through the water-air heat exchanger is blown by the fan. The fan is controlled by a frequency converter. We use the XV105D frequency

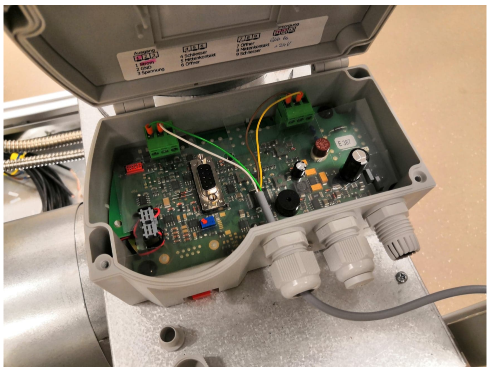


Figure 2.10: Pressure transducer PTSXR

converter from Dixell in Fig. 2.11. the frequency converter is controlled by an analog signal 0-10V. The frequency inverter is powered by 230V AC.

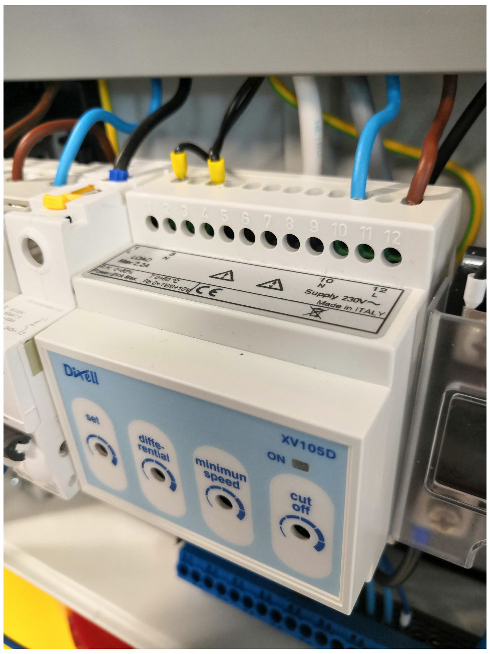


Figure 2.11: Frequency converter XV105D

2.2.5 Digital electricity meter

The power supply of the testbed is connected via an electricity meter from inepro Metering in Fig. 2.12. We use type PRO1-Mod electricity meter. The maximum load of the electricity meter is 45A at 230V AC. To connect electricity meter we use RS485 with Modbus protocol.

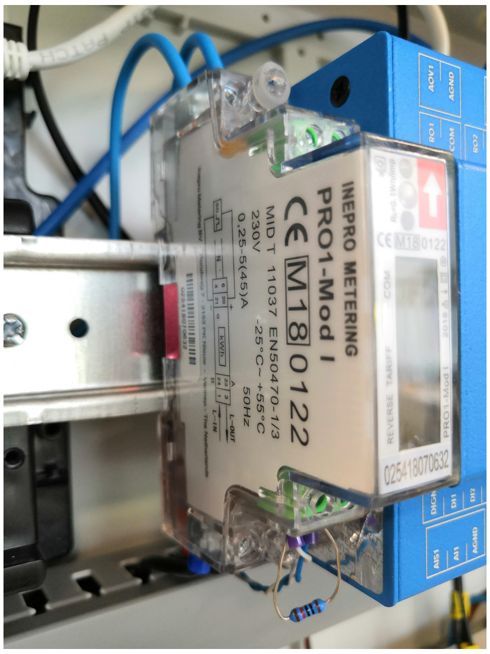


Figure 2.12: PRO1-Mod electricity meter

2.2.6 SSR relay

We use SSR relay to control the power of the heating coil in the boiler. The SSR relay is controlled by the controller's digital output in PWM mode.

Used relay is KSD215AC8 from COSMO in Fig. 2.13. The maximum current load of the SSR relay is 15A at 600V AC. This relay must be mounted on the passive cooler otherwise it may be destroyed.

2.2.7 Boiler

For heating water, we use the water heater from Dražice in Fig. 2.14. We use SSR relay to control the power of the heating coil. The water heater is powered by 230V AC.

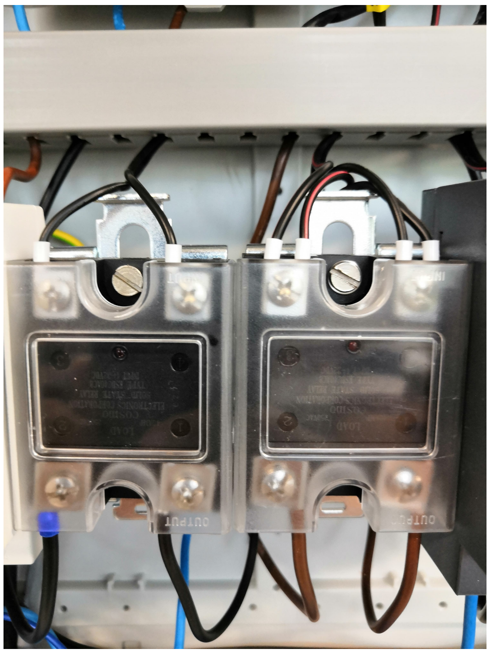


Figure 2.13: SSR relays for boiler

Figure 2.14: Water heater

2.2.8 Water pump

Primary flow through a water heater is induced by a primary pump from Grundfos MAGNA 3 in Fig. 2.15. To connect this pump we use RS485 with Modbus protocol. The pump is powered by 230V AC.

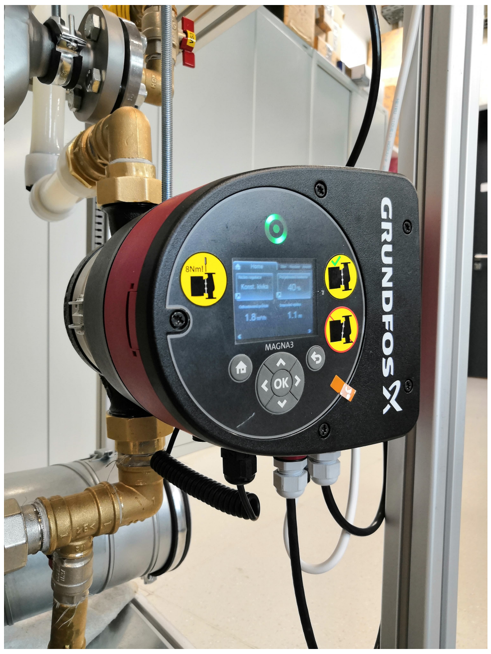


Figure 2.15: Grundfos MAGNA 3

2.2.9 Touchscreen

The testbed has a touch display that displays water temperature, air temperature, water flow and some more information. The display is from Weintek and its type is MT8102iP in Fig. 2.16. To connect display we use Ethernet with Modbus RTU over TCP protocol. The display is powered by 24V DC.

2.2.10 Electronic valve

An electronic flow control valve is mounted on the secondary circuit to heat the water through the exchanger. It is a N05010-SW2 valve from Honeywell in Fig. 2.17. The control is analog with voltage level in the range of 0-10V. The valve is powered by 24V DC.

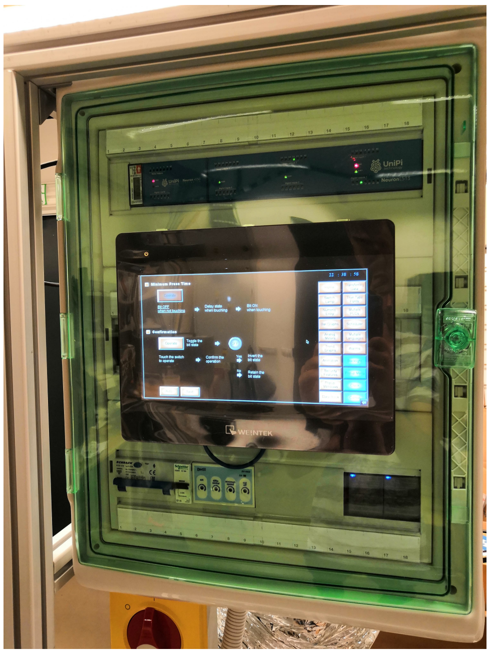


Figure 2.16: Touchscreen

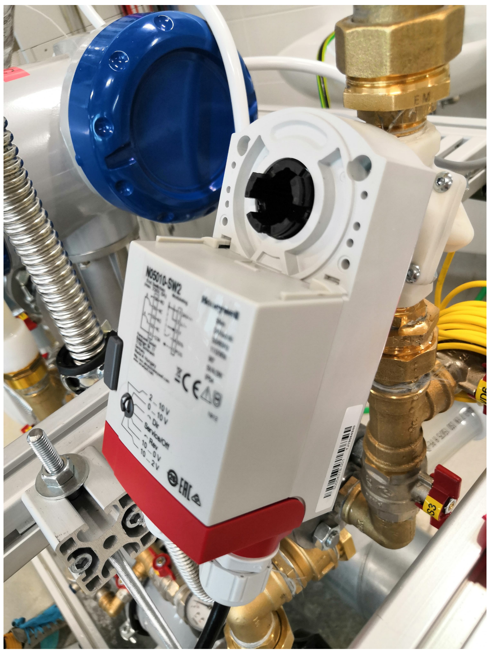


Figure 2.17: Electronic valve

2.3 Control unit

We have to communicate with each testbed component. Some components are passive, such as thermometers, and other active, such as pump or fan. Thus we need to have some control unit. In this section, we summarize the requirements for the control unit and compare them with possible candidates.

2.3.1 Requirements

A suitable control unit must have digital inputs and outputs for PWM, analog outputs for voltage generation, analog inputs for measuring voltage, resistance and the current loop. An additional required interface is an RS485 bus and ethernet. In addition, the hardware must be able to run custom software in parallel threads and has to be powerful enough to enable optimization solvers to operate on the control unit.

2.3.2 Possible control units

One of the candidates was the ATmega microcontroller on the Arduino prototyping board. However, this chip does not have an analog input for the current loop, RS485 bus, or Ethernet port. We must use additional electronics to read current loop or resistance sensor and some extension component for ethernet and RS485 interface.

Other alternatives included a microcontroller from STMicroelectronics. This chip has an ethernet and RS485 interface but does not have an analog input for the current loop. We must use additional electronics to read the current loop and resistance sensor.

Other alternatives included programmable logic controllers (PLC). We wanted to be able to access the inputs and outputs with full control. Another selection criterion for PLC is the ability to run custom software. Now we want control the testbed remotely using the Matlab Simulink application, but in the future we will want to move the control application to the PLC. So we compared PLCs from TECO [4], Beckhoff [5] and UniPi [6]. We have decided for UniPi because it is possible to access to operating system and run custom applications.

UniPi PLC is based on Raspberry Pi with an expansion board that handles input and output ports. The whole PLC is in an aluminum case, which can be mounted on a DIN rail in a rack. With this PLC, we have full-access Linux OS.

2.3.3 UniPi Neuron L513

UniPi Neuron L513 control unit was selected for testbed control. It is shown in Fig. 2.18. This control unit has several analog inputs for measuring voltage, current signals, and resistance. The analog outputs of this unit can provide voltage in range from 0 to 10V DC. There are also digital inputs with debounce circuit and digital outputs with mode pulse width modulation (PWM). There are also relay outputs for switching large current loads.

All UniPi products work on 24V DC, which is used by most peripherals so we can use one power supply to power the entire testbed.

All input and output ports can be controlled via the Evok API. We can communicate with the Evok API via ethernet. We can use several protocols, for example, Websocket, RESTful, JSON-RPC. In this project, we use an HTTP request with JSON data format. [7]

type	count	function
digital inputs	16	debounce
digital outputs	4	PWM, current load 750mA, maximum voltage 50V
relay outputs	10	current load 5A, maximum voltage 230V AC or 30V DC
analog inputs	9	measuring voltage 0-10V, current 0-20mA and resistance 0-1960kΩ
analog outputs	9	output voltage 0-10V
Ethernet	1	10/100Mbit
RS485	3	
1-Wire	1	

Table 2.1: Inputs and outputs on UniPi Neuron L513

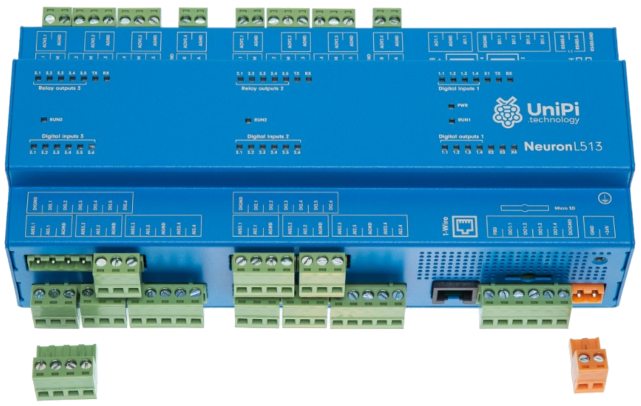


Figure 2.18: UniPi Neuron L513, source: https://unipi.technology

2.3.4 UniPi Neuron Extension xS50

UniPi Neuron Extension xS50 is an extension unit for Neuron controller units allowing extend inputs and outputs port. It is shown in Fig. 2.19. The extension unit is connected via an RS485 bus with protocol Modbus. The extension unit has its own processor, which communicates with a master control unit. [8]

type	count	function
digital inputs	6	debounce
relay outputs	5	current load 5A, maximum voltage 230V AC or 30V DC
analog inputs	4	measuring voltage 0-10V, current 0-20mA and resistance 0-1960kΩ
analog outputs	4	output voltage 0-10V
RS485	1	

Table 2.2: Inputs and outputs on UniPi Neuron L513

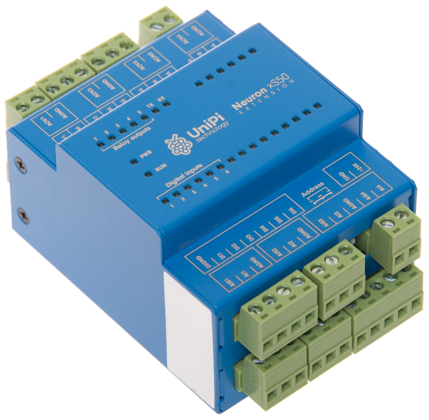


Figure 2.19: UniPi Neuron extension xS50, source: https://unipi.technology

Chapter 3

Software

Before programming, it was necessary to design software architecture. My job was to design software architecture and program a server application that communicates with clients and handles inputs and outputs ports. In the next step I created a block for Simulink in Matlab that communicates with the server application. The result is the control of the testbed from the application in Simulink in Matlab.

3.1 Architecture

At the beginning I had to find out the software requirements. Several applications will run on the PLC. One of the applications is EVOK, which control input and output ports. Another application is any database that is used to record all activity. In the future will run on the PLC predictive control applications, so this requirement must be respected. The software architecture design is shown in Fig 3.1. Other requirements related to the server application, which provides communication between programs.

UniPi is based on operation system Raspbian with service EVOK, which control input and output ports. On this operating system we can run custom software. My server application runs on Raspbian and communicate with EVOK. The RS 485 serial line is controlled directly from my application. My server application also communicates with the client over ethernet.

My server applicatin is multithreaded. The application provides communication with clients over Ethernet, checking client syntax and communicating with EVOK via API over Ethernet. At the same time, the application must check that the security rules are followed. Security rules are specified when configuring peripherals.

3.1.1 Requirements

Communication server application requirements are:

- easy peripheral configuration,
- flexible communication over ethernet with clients,
- implementation of security rules.

Another software application requirements are OOP design for easy extension of other types of peripherals, stable and fast communication, recalculating physically measured units into human readable units.

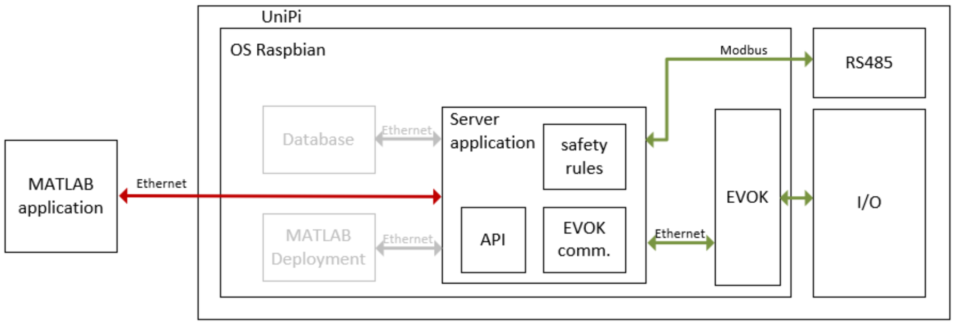


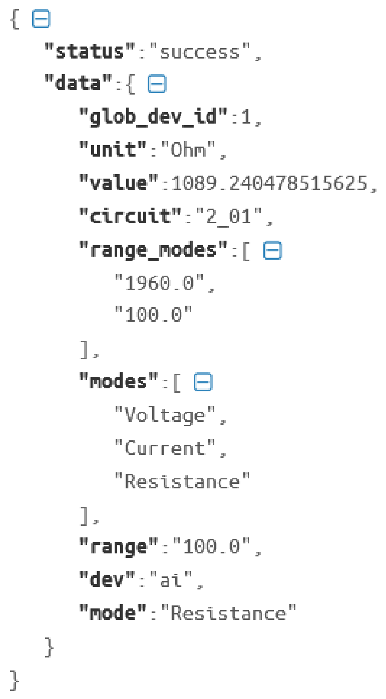
Figure 3.1: Software architecture

3.2 Server

My server application implements a flexible communication stack for all provided periphetals. At the same time, the application checks the rules to keep the testbed in safe state.

3.2.1 Communication protocol

For communication with EVOK I chose JSON API over HTTP requests. I had to study the JSON API documentation for EVOK. Each port have own identificator, which consists of a group identifier and a port identifier. For example, if you want get analog value from analog port number one on second logic group, you muset use get http request on address `http://unipi-ip-addr:8080/json/analoginput/2_01`. EVOK returns data in JSON format, if field with name `status` has value `success`, you can read field `data`. In field data from HTTP response you find field with required value. [9]



```
{
   "status":"success",
   "data":{
      "glob_dev_id":1,
      "unit":"Ohm",
      "value":1089.240478515625,
      "circuit":"2_01",
      "range_modes":[
         "1960.0",
         "100.0"
      ],
      "modes":[
         "Voltage",
         "Current",
         "Resistance"
      ],
      "range":"100.0",
      "dev":"ai",
      "mode":"Resistance"
   }
}
```

Figure 3.2: EVOK response on request.

My server application allows you to set up all input and output ports on UniPi through EVOK. Analog input ports must be set to mode (voltage, current, resistance) according to what we want to measure. We must also specify range of measure value. Analog output ports can only operate in voltage mode with range 0-10V. Digital input ports can be connected up to 50V DC. Digital output ports can operate in simple or pulse width modulation mode. The RS485 serial line is operated directly from my server application.

I used a Websocket protocol to communicate between the clients and the PLC. Data between the client and the server are send in JSON format. The client requests peripherals information from the server not ports information.

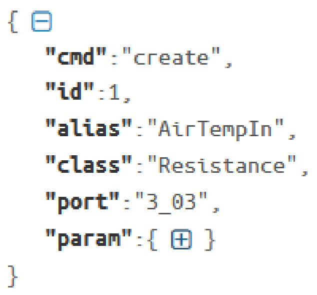
Each of the peripherals has a its unique id, that was specified in configuration. Peripherals return values recalculated in human readable unit instead of raw data. The recalculation formula is given in the peripheral configuration. For example, the thermometer does not return the value of its resistance, but the temperature in degrees Celsius.

My communication API include these types of messages:

- create - create a periphery
- remove - remove a periphery by unique id
- clear - remove all periphery
- set - set values or parameters of peripheral
- get - get values and parameters of peripheral
- state - get values and parameters of all peripherals

Create periphery

Each peripheral must be configured after the physical connection. For configuration, we use a message with the structure shown in Fig. 3.3. The values for each field are listed in the table 3.1. A specific example of thermometer and Modbus device configuration is shown in Fig. 3.4.



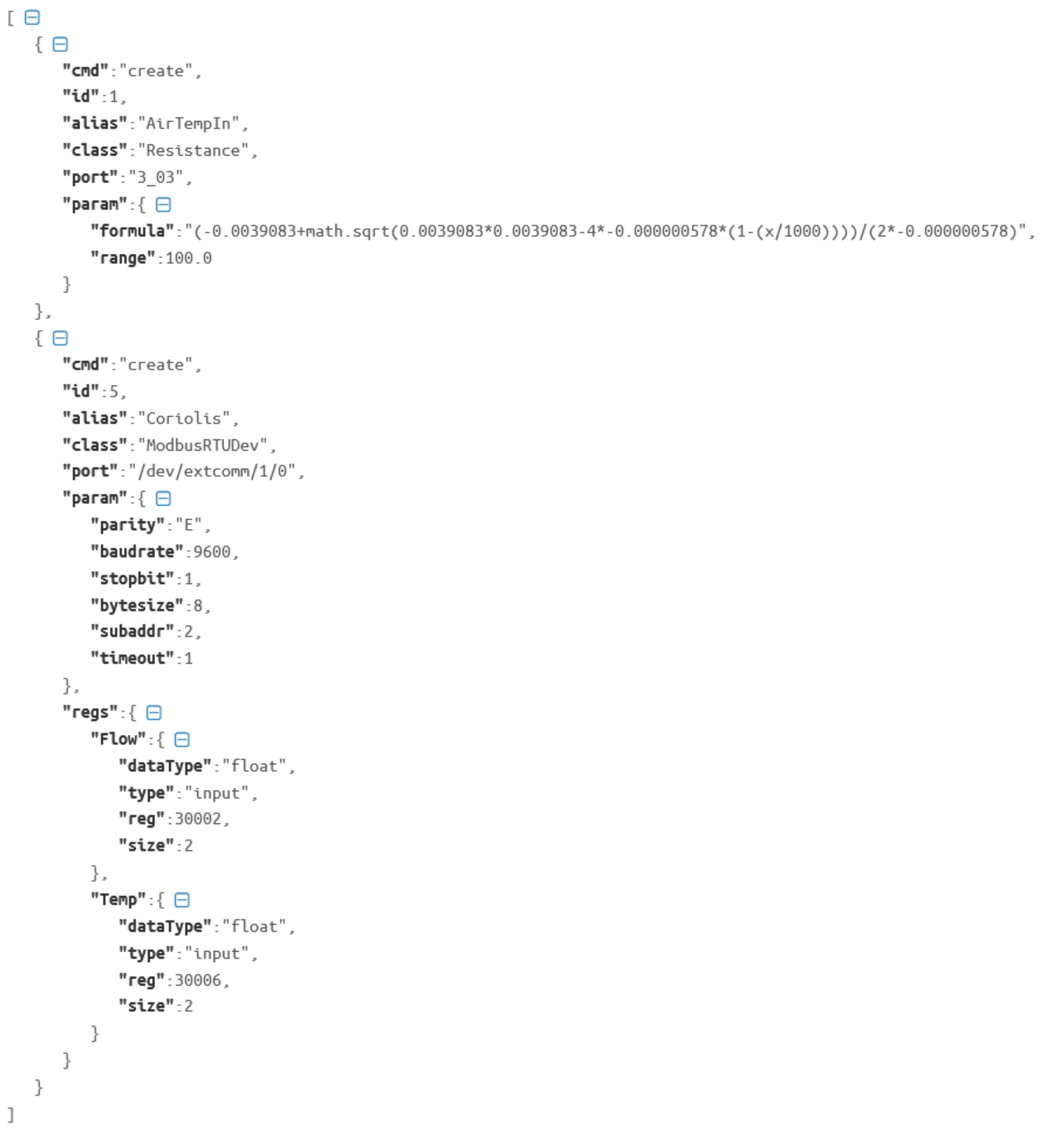
```
{
   "cmd":"create",
   "id":1,
   "alias":"AirTempIn",
   "class":"Resistance",
   "port":"3_03",
   "param":{ }
}
```

Figure 3.3: Create periphery command structure.

In addition, the Modbus device has a list of registers that we specify in the `regs` field. In this list is named fields with register parameters. Each item in registry configuration is listed in the table 3.2.

The peripheral port specification varies according to the peripheral type. Peripherals connected via RS485 have a port in the format `/dev/extcomm/<number_of_connector>/0`. Other peripherals have port by physical connector number on UniPi, for example analog input AI2.2 is assigned port 2_02.

The server response to this command is success or error. If the answer is an error, then we get the error information in the `msg` field. The most common mistake is to enter an already used ID or a bad port identifier.



```
[
   {
      "cmd":"create",
      "id":1,
      "alias":"AirTempIn",
      "class":"Resistance",
      "port":"3_03",
      "param":{
         "formula":"(-0.0039083+math.sqrt(0.0039083*0.0039083-4*-0.000000578*(1-(x/1000))))/(2*-0.000000578)",
         "range":100.0
      }
   },
   {
      "cmd":"create",
      "id":5,
      "alias":"Coriolis",
      "class":"ModbusRTUDev",
      "port":"/dev/extcomm/1/0",
      "param":{
         "parity":"E",
         "baudrate":9600,
         "stopbit":1,
         "bytesize":8,
         "subaddr":2,
         "timeout":1
      },
      "regs":{
         "Flow":{
            "dataType":"float",
            "type":"input",
            "reg":30002,
            "size":2
         },
         "Temp":{
            "dataType":"float",
            "type":"input",
            "reg":30006,
            "size":2
         }
      }
   }
]
```

Figure 3.4: Thermometer and flowmeter configuration.

type	field	value
for all	id alias port class param regs	own unique identifier own name of periphery for human readability UniPi physical port specify type of periphery (`Voltage`, `Current`, `Resistance`, `Simple`, `PWM`, `Relay`, `ModbusRTUDev`) below list of modbus registres (only for Modbus device) listed in the table 3.2
field `param` for Voltage inputs	direction range formula	direction of port (`input`) maximum value (`10`) V formula of variable x for recalculate
field `param` for Voltage outputs	direction formula	direction of port (`output`) formula of variable x for recalculate
field `param` for Current inputs	direction formula	direction of port (`input`) formula of variable x for recalculate
field `param` for Current outputs	direction range formula	direction of port (`input`) maximum value (`100, 1960`) kOhm formula of variable x for recalculate
field `param` for Simple inputs	direction	direction of port (`input`)
field `param` for Simple outputs	direction	direction of port (`output`)
field `param` for Relay outputs		
field `param` for PWM outputs	clock	switching frequency (`0.1 - 48000000`)
field `param` for Modbus	subaddr baudrate parity stopbit timeout bytesize	Modbus device address (`1 - 127`) Modbus device baudrate (`1200, 2400, 4800, 9600, 115200, etc.`) Modbus device parity (`'E','O','N'`) Modbus device stopbit (`'1','2'`) Response timeout Byte size in bits `8`

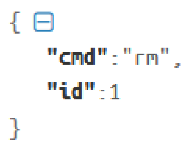
Table 3.1: Required fields for periphery configuration.

field	value
reg	Register address
size	Value size in bytes
dataType	Value datatype (`'float'`,`'int'`,`'string'`)
type	Register type (`'holding'`, `'input'`)

Table 3.2: Required register parameters for Modbus device.

Remove periphery

To remove a periphery, use the command with the structure shown in Fig. 3.5. This command only has a parameter with the peripheral ID value to be removed from the configuration. The server response to this command is success or error. If there is no periery with the specified ID, we get an error response.

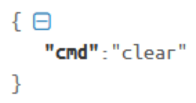


```
{
   "cmd":"rm",
   "id":1
}
```

Figure 3.5: Remove periphery with ID 1.

Remove all periphery

To remove all peripherals, use the command with the structure shown in Fig. 3.6. This command has no other parameters. The server response to this command is always a success.



```
{
   "cmd":"clear"
}
```

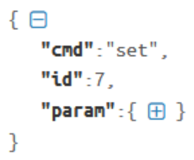
Figure 3.6: Remove all peripherals.

Set periphery parameters

To set periphery parameters, use the command with the structure shown in Fig. 3.7. This command has a mandatory ID and `param` field. The values for each type of periphery are listed in the table 3.3.

The server response to this command is success or error. If the answer is an error, then we get the error information in the `msg` field. The most common error is the wrong range of set parameters.

For Modbus devices, the possibility to change parameters and registry values via API is not implemented at this time.



```
{ ⊟
   "cmd":"set",
   "id":7,
   "param":{ ⊞ }
}
```

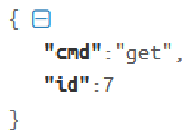
Figure 3.7: Set peripheral parameters.

type	field	value
for all	id param	periphery unique identifier below
field `param` for Voltage outputs	value	values of voltage from range
field `param` for Simple outputs	state	state of output `0,1`
field `param` for Relay outputs	state	state of output `0,1`
field `param` for PWM outputs	pwm_duty	state of output `0 - 100`
field `param` for PWM outputs	pwm_duty	state of output `0 - 100`

Table 3.3: Required fields for change periphery parameters.

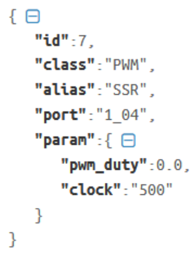
Get parameters of one periphery

This command is used to find available peripheral information that is specified by the `id` field value. An example of this command is shown in Fig. 3.8. The server response to this command is success or error. Example of success response is shown in Fig. 3.9. If the answer is an error, then we get the error information in the `msg` field. The most common error is a non-existent peripheral id.



```
{
   "cmd":"get",
   "id":7
}
```

Figure 3.8: Get peripheral informations.

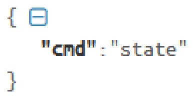


```
{
   "id":7,
   "class":"PWM",
   "alias":"SSR",
   "port":"1_04",
   "param":{
      "pwm_duty":0.0,
      "clock":"500"
   }
}
```

Figure 3.9: Response from server with informations.

Get parameters of all peripheral

Use this command to get the parameters of all configured peripherals that are contained in the testbed. An example of this command is shown in Fig. 3.10. The server response is an array that contains data about all peripherals with the same structure as a response to the status query of one peripheral.

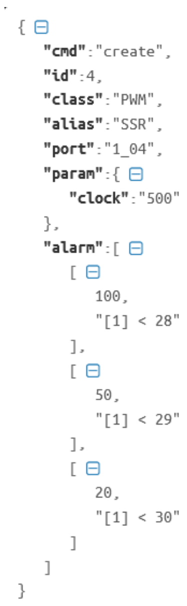


```
{
   "cmd":"state"
}
```

Figure 3.10: Get peripheral informations.

3.2.2 Safety rules

If you want use safety rules, for example if water temperature exceeds 90 °C, the boiler switches off, you can specify them in peripheral configuration. Safety rules specify in `alarm` field as array of pair value and string. Each item in array has equation where actual periphery value is represented as its unique identifier in square brackets, e.g. ”[1]”, it is also possible to use comparsion operators and constants. For Modbus devices, use the alarm field for each register separately. Example of SSR Relay with id 7 alarm configuration based on temperature from thermometer with id 1 is shown in Fig. 3.11.



```
{
  "cmd":"create",
  "id":4,
  "class":"PWM",
  "alias":"SSR",
  "port":"1_04",
  "param":{
    "clock":"500"
  },
  "alarm":[
    [
      100,
      "[1] < 28"
    ],
    [
      50,
      "[1] < 29"
    ],
    [
      20,
      "[1] < 30"
    ]
  ]
}
```

Figure 3.11: Example of configuration of alarm for periphery with id 7.

3.2.3 Server application

I chose scripting programming language Python for programming my server application. At the beginning of my work I only tested communication with EVOK through various APIs, finally I chose JSON API communication. I made a data structure design to hold peripheral information. Data structure design uses objects. Each object represents a different type of periphery. Object design uses inheritance. Object instances represent specific peripherals and are stored in the collection. Each object has methods for create JSON data format output, get or set value or registres and get recalculate value. The server application consists of some parts. The structure of the object is shown in Fig. 3.12.

The application uses libraries:

- threading - standard library
- time - standard library
- json - standard library
- requests - require install (`pip install requests`)
    - https://github.com/kennethreitz/requests

- websocket_server - require install (`pip install websocket-server`)
    - https://github.com/Pithikos/python-websocket-server
- pymodbus - require install (`pip install pymodbus`)
    - https://github.com/riptideio/pymodbus

API

Using the API, the client application communicates with the server. In this case, the Websocket protocol is used for communication. I chose this protocol because I used it when communicating with EVOK and wanted to keep the same communication protocol. Communication is through the messages I have described in the previous section. Each received message is executed and the response is sent back to the client. JSON data format is used for message representation.

Safety rules

This part of the server checks client requests and compares them with defined security rules. If the client request violate security rules, then the client request is discarded. For example, if the client asked to turn on the boiler, but the water would be overheated.

Communication with EVOK

For communication with EVOK I chose JSON API over HTTP requests. This part of the application sends commands to EVOK and receives data from EVOK.

Modbus RTU on RS485

I used Python library pymodbus [10] to control the RS485 bus with the Modbus protocol [11].

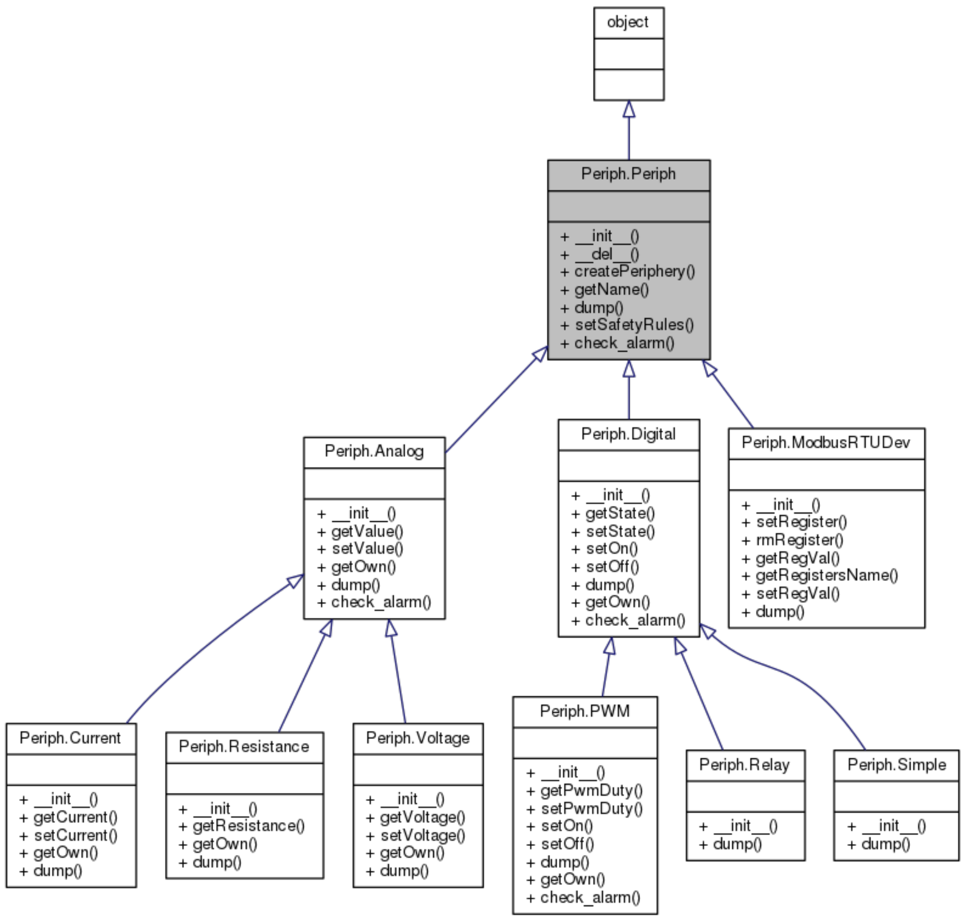


Figure 3.12: Inheritance diagram.

3.2.4 Set application as a service on Linux

We want the server application to run after the operating system starts. So we need to configure the python application as a service. There are many ways to solve this problem. I chose the `systemd` method. This method to run a program on UniPi at startup is to use the `systemd` files. Systemd provides a standard process for controlling what programs run when a Linux system boots up. Note that `systemd` is available only from the Jessie versions of Raspbian OS.

Step 1, create a unit file. I created a new file in `/lib/systemd/system/` with name `hydro.service`. Put into file content that defines a new service and its parameters. Set file privileges to `644`. The content of the file are shown in Fig. 3.13.

```
[Unit]
Description=Server app
After=multi-user.target

[Service]
Type=idle
ExecStart=//usr/bin/python /home/pi/PyApp/server.py

[Install]
WantedBy=multi-user.target
```

Figure 3.13: File hydro.service content.

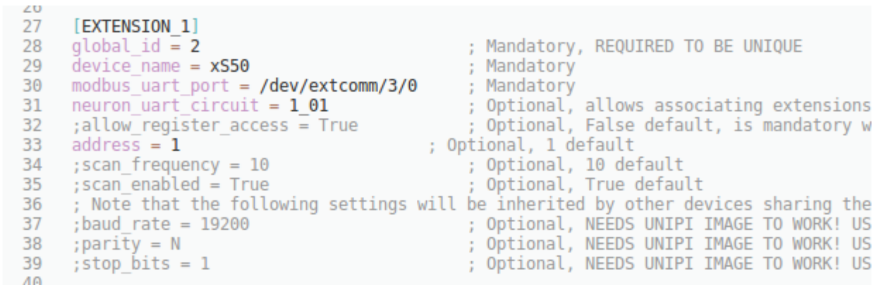
Step 2, configure `systemd`. We must tell `systemd` to start start it during the boot sequence. We use command: `systemctl daemon-reload` and `systemctl enable hydro.service`. [12]

Now we have configure server application as service and this application automatically start at each operating system start up.

3.2.5 Configuration extension unit in EVOK

For the functional connection of UniPi Neuron L513 and UniPi Neuron Extension xS50, it is necessary to connect the RS485 bus connectors and activate the built-in terminators using the switch at the end of the bus.

It is necessary to configure the hardware in the main unit with which extension unit and on which line to communicate. We configure extension unit in the `evok.conf` file in the `/etc` directory. A sample of the file content is shown in Fig. 3.14. Here we set the global module ID, the file name, which contains information about the MODBUS registers, port and address. Other parameters are optional. All the parameters are described in the configuration file comments. This file may vary in different versions of the EVOK. [9]



```
[EXTENSION_1]
global_id = 2                               ; Mandatory, REQUIRED TO BE UNIQUE
device_name = xS50                          ; Mandatory
modbus_uart_port = /dev/extcomm/3/0         ; Mandatory
neuron_uart_circuit = 1_01                  ; Optional, allows associating extensions
;allow_register_access = True               ; Optional, False default, is mandatory w
address = 1                             ; Optional, 1 default
;scan_frequency = 10                        ; Optional, 10 default
;scan_enabled = True                        ; Optional, True default
; Note that the following settings will be inherited by other devices sharing the
;baud_rate = 19200                          ; Optional, NEEDS UNIPI IMAGE TO WORK! US
;parity = N                                 ; Optional, NEEDS UNIPI IMAGE TO WORK! US
;stop_bits = 1                              ; Optional, NEEDS UNIPI IMAGE TO WORK! US
```

Figure 3.14: Example of evok.conf file

3.3 Matlab application

The last part of my work is to create an application for Matlab Simulink. This application is used for simple testbed control and measurement. I started by learning about Matlab and creating a Simulink object. To create a communication link, I used a third-party library WSClient [13] that provides Websocket client. This library requires several additional packages to install and add a path in Matlab. The library needs Java to run.

The Matlab libraries that mediate the websocket client are more. I chose WSClient library with the help of my supervisor, it is the easiest to use and has the least dependency on other packages.

After the connection has been established successfully, it is necessary to process messages receive from the server and send control instructions to the server. For this I have to use the functions to parse JSON messages. Simulink object works in steps. At each step, the input values are read and processed and sent in JSON form messages to the server. Incoming messages with peripheral status information are processed and values passed to the simulink object output. Both input and output values are represented as a column vector. Simulink object output data is displayed using part that contain a Simulink library. An example of the interface in Simulink is shown in Fig 3.15.

Each time the connection is established, the Matlab application sends a command to turn on the peripheral power to make measurements. It is the command to activate the digital output on which the relay is connected. At the end of the connection the digital output is deactivated and the peripherals are switched off.

When testing the application and the server, the data shown in Fig 3.16 was measured. At the time of testing, there is still no functional water heating. You can see in the graph that the fan started in 35 seconds, which increased the airflow and the current consumption of the testbed. The air flow is measured using an AL-grid and a pressure transducer. Electricity consumption is measured using an electricity meter.

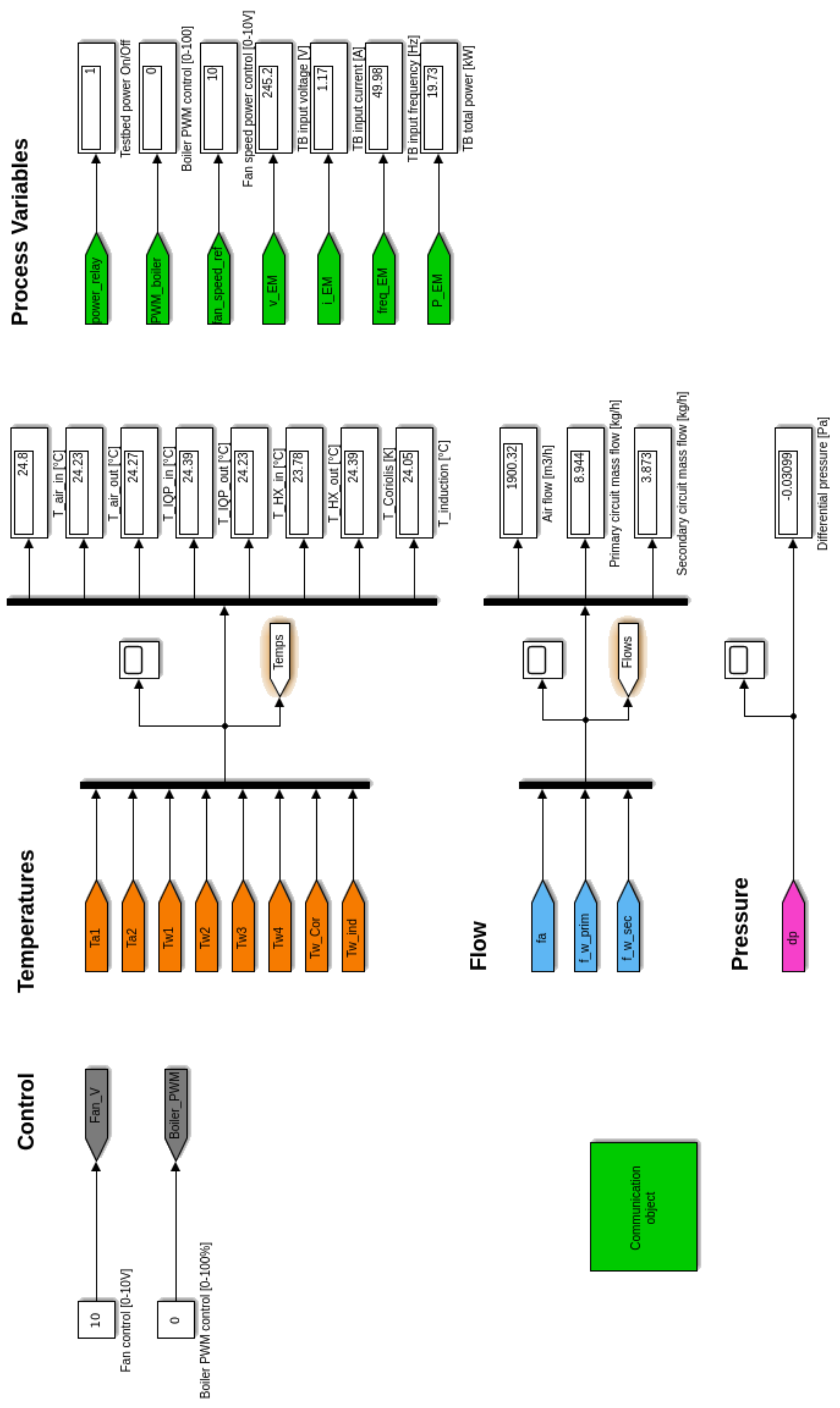


Figure 3.15: Data representation in Simulink.

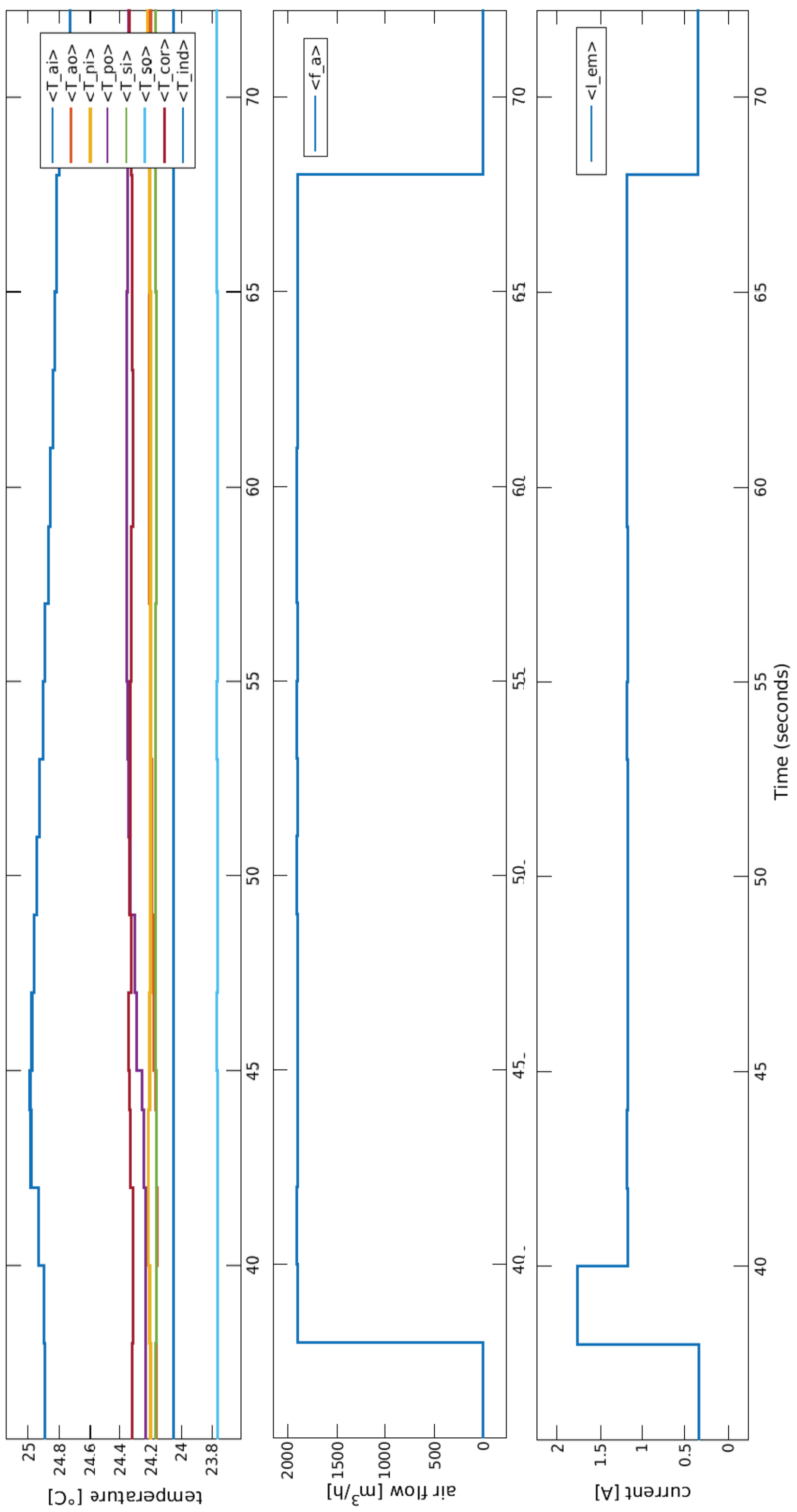


Figure 3.16: Output graph of measured values.

3.4 Touch screen communication

At the beginning I had to create a project in EasyBuilder Pro. This program is free from Weintek's display manufacturer. In the project I had to define that the display is connected via Ethernet and mediates Modbus server, which is used for communication with UniPi PLC. In a graphical environment, I created a screen that displays temperature information. There is also a switch on the display that can be switched by touch. The displayed temperature value is read from the defined Modbus server register. The switch state is also stored in the Modbus server register.

I created a demonstration program in Python that connects to the display as well as any Modbus device. This program reads the value from the thermometer on analog input of UniPi and it writes to the appropriate register on the Modbus server display. The program also reads the Modbus server registry value where the graphical switch status is stored. According to the value in the register, the program instructs the activation or deactivation of the digital output UniPi.

The demo application results can be seen in Fig 3.17 and 3.18.

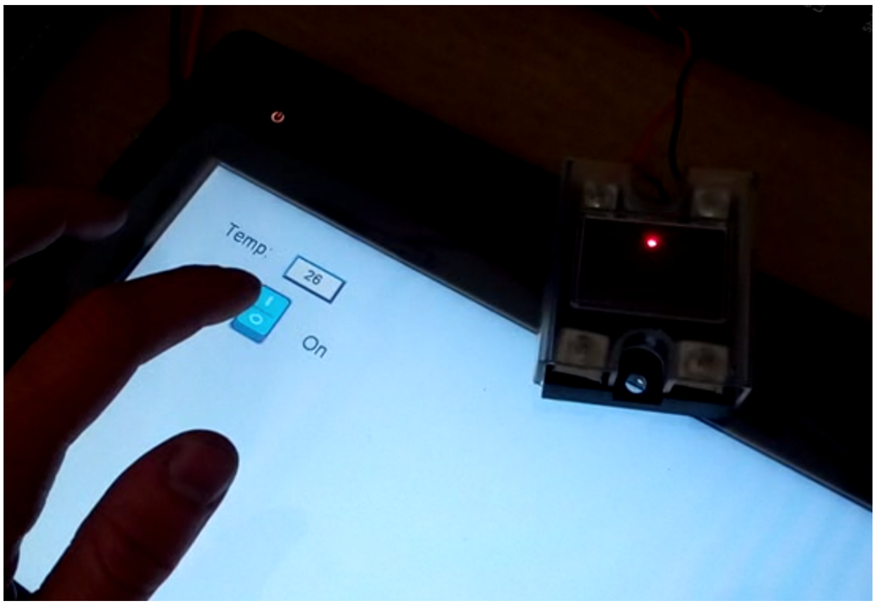


Figure 3.17: Example of relay on.

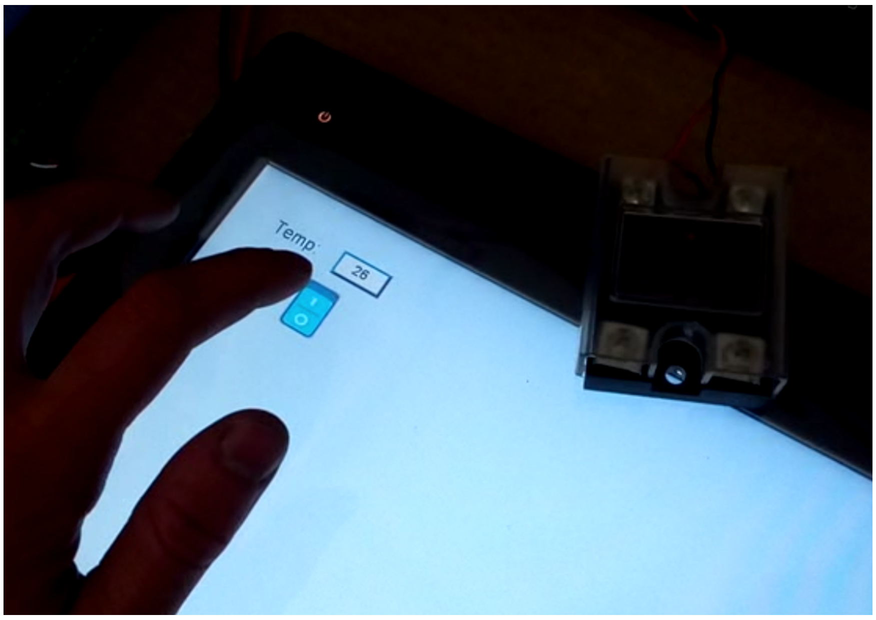


Figure 3.18: Example of relay off.

Chapter 4

Conclusion

The aim of the work was to study building automation systems, to get acquainted with the hydraulic part of the test bed and to design the electrical part including necessary components. The electrical part included the selection of a PLC that communicates with peripherals. This part of the work has been done.

Another aim of the work was to program a communication stack for all peripherals. The automation system software should be flexible and in terms of peripheral changes. A security task must be running on the PLC to monitor the safe status of the testbed. This part of the work has been done.

The last aim of the work was to prepare a SCADA operating on a graphical touchscreen. This part of the work has been partially done. I created a way to communicate between display and UniPi and created a demo application to show the function.

Future work on the project will include changing the communication protocol between PLC and clients and API extensions to writes to the Modbus registers and communicate with Modbus device over Ethernet. The currently used Websocket protocol is unsatisfactory. I want to change the protocol so that only one common is used for all Ethernet communication. The reason for this change is the error that occurs when communicating between the server and the client. I would like to add BACnet communication. [14]

Bibliography

[1] Zelenka, D.: Cyber-physical One-pipe Hydronic Heating Testbed. *CTU*, 2019.

[2] Dostál, J.: Studie proveditelnosti OPPPR. *Koncept Praha*, 2018.

[3] OPC Foundation: *What is OPC? [online]*. [cit. 2019-05-20]. Accessed from: `https://opcfoundation.org/about/what-is-opc/`

[4] TECO: *PLC Tecomat Foxtrot [online]*. [cit. 2019-05-20]. Accessed from: `https://www.tecomat.cz/products/cat/cz/plc-tecomat-foxtrot-3/`

[5] Beckhoff: *Beckhoff PLC [online]*. [cit. 2019-05-20]. Accessed from: `https://www.beckhoff.com/`

[6] UniPi.technology: *UniPi Neuron [online]*. [cit. 2019-05-20]. Accessed from: `https://www.unipi.technology/products/unipi-neuron-3?categoryId=2&categorySlug=unipi-neuron`

[7] UniPi.technology: *UniPi Neuron L513 [online]*. [cit. 2019-05-20]. Accessed from: `https://www.unipi.technology/unipi-neuron-l513-p106?categoryId=2`

[8] UniPi.technology: *UniPi Neuron xS50 [online]*. [cit. 2019-05-20]. Accessed from: `https://www.unipi.technology/unipi-neuron-extension-xs50-p111?categoryId=13`

[9] UniPi.technology: *EVOK - the UniPi API [online]*. [cit. 2019-05-20]. Accessed from: `https://evok.api-docs.io/1.0/jKcTKe5aRBCNjt8Az/introduction`

[10] Read the Docs: *PyModbus - A Python Modbus Stack [online]*. [cit. 2019-05-20]. Accessed from: `https://pymodbus.readthedocs.io/en/latest/readme.html`

[11] Chipkin, P.: *Modbus For Field Technicians*. CreateSpace Independent Publishing Platform, 2011, ISBN 978-1456376444.

[12] DigitalOcean: *How To Use Systemctl to Manage Systemd Services and Units [online]*. [cit. 2019-05-20]. Accessed from: `https://www.digitalocean.com/community/tutorials/how-to-use-systemctl-to-manage-systemd-services-and-units`

[13] tbxManager: *Websocket client for Matlab [online]*. [cit. 2019-05-20]. Accessed from: `http://www.tbxmanager.com/package/view/wsclient`

[14] Domingues, P.: Building automation systems: Concepts and technology review. *Computer Standards Interfaces*, 2016.

[15] Samad, T.: System architecture for process automation: Review and trends. *Journal of Process Control*, 2007.

[16] Zlevor, O.: Heat flow control of water-to-air heat exchanger. *CTU*, 2017.

Appendix A

CD Content

A CD is attached to the printed version of this thesis. It contains an electronic version of this thesis.