České učení technické v Praze
Fakulta strojní
Ústav přístrojové a řídící techniky

Simulátor výrobního zařízení založený na RPi+UniPi

Bakalářská práce

Kirill Rassudikhin

Bakalářský program: Teoretický základ strojního inženýrství
Bakalářský obor: Bez oborový
Vedoucí práce: Mgr. Ing. Jakub Jura, Ph.D.

Praha, srpen 2020

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Rassudikhin** Jméno: **Kirill** Osobní číslo: **466560**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Teoretický základ strojního inženýrství**

Studijní obor: **bez oboru**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Simulátor výrobního zařízení založený na RPi+UniPi

Název bakalářské práce anglicky:

Production system simulator

Pokyny pro vypracování:

1) Seznamte se s programováním RPi
2) Připojte k RPi modul vstupů a výstupů UniPi
3) Navrhněte a vyzkoušejte způsob simulace vybraných technologických prvků pomocí několika platforem (například Python, Mervis ...)
4) Vyzkoušejte funkčnost simulátoru spojením s řídícím PLC

Seznam doporučené literatury:

[1] MARTINÁSKOVÁ, Marie a Ladislav ŠMEJKAL. Řízení programovatelnými automaty. Vyd. 2. V Praze: Vydavatelství ČVUT, 2004. ISBN 8001029255.
[2] Unipi, firemní dokumentace, https://www.unipi.technology/cs/
[3] Python tutorial, https://www.tutorialspoint.com/python/index.htm
[4] Mervis, firemní dokumentace, https://mervis.info/#/en/home
[5] Mervis, firemní dokumentace Unipi, ,https://kb.unipi.technology/en:sw:01-mervis:06-tutorials

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Mgr. Jakub Jura, Ph.D., U12110.3

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

Datum zadání bakalářské práce: **30.04.2020** Termín odevzdání bakalářské práce: **07.08.2020**

Platnost zadání bakalářské práce: ____________

[illegible] [illegible] [illegible]

Ing. Mgr. Jakub Jura, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Michael Valášek, DrSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

14.07.2020
Datum převzetí zadání

[illegible]
Podpis studenta

Vedoucí práce:
Mgr. Ing. Jakub Jura, Ph.D.
Ústav přístrojové a řídící techniky
Fakulta strojní
České vysoké učení technické v Praze
Technická 2
160 00 Praha 6
Česká republika

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, s tím, že její výsledky mohou být dále použity podle uvážení vedoucího diplomové práce jako jejího spoluautora. Souhlasím také s případnou publikací výsledků diplomové práce nebo její podstatné části, pokud bude uveden jako její spoluautor.

V Praze 2020

....
Kirill Rassudikhin

Abstract

Abstract

The main goal of this bachelor thesis is to develop a hardware simulator of specific automation components (with method of digital twin). The simulator should make developing control programs for PLC easier. The simulator in the current version simulates a system of the linear pneumatic actuator with a monostable and bistable direct control valve. The simulator was built on the basis of Raspberry Pi B+ v 1.2 and UniPi v1.1. During development, several versions of the simulator were tested. The first version was strictly simulator without visualization or plotting. The second version included visualization, the third version included visualization and plotting. The theoretical part of this bachelor thesis describes real systems with monostable and bistable valves as an object for simulation. Those systems were translated into product system simulation models. Also theoretical part of the bachelor thesis describes hardware for simulation such as Raspberry Pi and UniPi, hardware for testing the simulator (PLC), and methods of communication between that hardware equipment. The practical part of the bachelor thesis describes the physical connection (electrical circuit) between hardware, programs for PLC, created for testing. But the major part of the practical part is the process of developing the simulator. It describes developing the simulator on three different platforms (Python, Node-Red, Mervis), used modules and methods, final programs, source codes, and also describes each platform.

Keywords: Raspberry Pi, UniPi, Product system simulator, Python, Node Red, Mervis, PLC, digital twin

Abstrakt

Cílem této bakalářské práce je vývoj hardwarového simulátoru automatizačních komponent (na způsob digitálního dvojčete výrobního zařízení) za účelem usnadnění vývoje řídicího programu v PLC. Simulátor v současné verzi modeluje chování lineárního pneumatického pohonu s bistabilmním a monostabilním rozvaděčem. Simulátor je hardwarově postaven na Raspberry Pi B+ v1.2 a UniPi v 1.1. Softwarově bylo otestováno několik verzí. První verze obsahovala jenom program simulátoru. Druhá verze také obsahovala vizualizaci. Třetí verze obsahovala vizualizaci a graf průběhu simulace. Teoretická část této bakalářské práce popisuje reálné systémy jako objekt simulace. Tyto systémy zatím budou převedené do modelu simulátoru výrobního zařízení. Takže teoretická část popisuje hadware použitý pro simulaci: Raspberry Pi B+ v1.2 a UniPi v.1.1, popisuje hardware pro testování (PLC) a metody komunikace mezi těmito zařízení. Praktická část bakalářky popisuje fyzické zapojení mezi jednotlivými hardware zařízení (elektronická schémata), programy pro PLC, vytvořený pro testování. Ale větší část je věnována procesu vyvíjení simulátoru, popisuje vyvíjení na třech různých platformách (Python, Node Red, Mervis), použité moduly a metody, výsledné programy, zdrojové kódy a jednotlivé platformy.

Klíčová slova: Raspberry Pi, UniPi, Simulátor výrobního zařízení, Python, Node

Red, Mervis, PLC, digitální dvojče

Poděkování

Děkuji mému vedoucímu Mgr. Ing. Ph.D. Jakubu Jurovi za rady a pomoc. Taky děkuji Ing. Matouši Cejnkovi za pomoc s pythonem a Ing. Ph.D. Marii Martináskově za rady ohledně pneumatiky.

List of Tables

List of Figures

Acronyms

DCV Direct Control Valve. 4–8, 30, 33–35, 40, 41, 43, 44, 56, 57

FBD Function Block Diagram. 21, 33, 37, 67

FPS Frames Per Second. 46, 48

GPIO General Purpouse Input/Output. 9, 10, 12, 14, 16–19, 24, 31, 38, 39, 41, 45, 68, 71

HMI Human Machine Interface. 70

I2C Inter Integrated Circuit. 12, 18–20, 24, 37–39, 66, 71

IDE Integrated Development Environment. x, xiii, 33, 37, 67–71

JSON Java Script Object Notation. 59

LD Ladder Diagram. 21, 22, 33

LED Light Emitting Diode. 12, 13

OS Operation System. 9, 67

PC Personal Computer. xiii, 36, 44, 45, 52

PLC Programmable Logic Controller. 1–3, 5, 7, 12, 13, 21–24, 26–28, 30, 31, 33, 34, 61, 71, 72

RPi Raspberry Pi. 5, 7, 9, 12, 16, 24, 38, 39, 46, 52, 59, 64, 66, 71, 73

SFC Sequential Function Chart. 21, 41, 56

SPI Serial Peripheral Interface. 18, 19

SSH Secure Shell. 9

ST Structured Text. 21, 33, 37, 67

Contents

Chapter 1

Introduction

Programming of control systems and it's connection to real field level technological equipment creates technological, economical and safety risks. The opportunity to test a control system with a certain program before connecting it to the real system is a significant lowering of those risks. Also, the opportunity to do this without expensive technology is a significant lowering of the price of the development and testing of a certain system for a certain technological process. So, developing a simulator with open source or low-cost technology for testing controlling programs is the main theme of this bachelor thesis.

Keeping in mind low-cost technology made me choose open source platforms for developing a simulator. Those platforms are Python, Node-Red, and Mervis. The last one is not an open-source platform, but it comes in a box with hardware, which was chosen for developing the simulator. Raspberry Pi as hardware perfectly meets the requirements with low cost, wildly available technology. Anyone in almost any country could buy Raspberry Pi and use a simulator, developed for it. Problem is that Raspberry Pi doesn't work with high industrial voltages, that's why it needed enhancement for this task. The enhancement is extension board UniPi v1.1 with ports, configured for industrial 24 V, and with a set of relays.

In Introduction I should explain that automation systems are divided on five different levels of automation: field-level automation, control-level automation, and enterprise or information-level automation also there are Planning Level and highest level - The Management Decision Making Layer.This is called pyramid of automation, the scheme of pyramid can be seen on figure 1.1. Field-level automation is the lowest level of the automation hierarchy and consists of field devices such as sensors and actuators. Sensors, the eyes and ears of automation, collect data on temperature, pressure, speeds, feeds, and so on, convert it to electrical signals, and relay it up to the next level - control level automation. This level consists of various automation controllers such as PLC controllers that gather process parameters from various sensors. The automatic controllers then drive the actuators based on the processed sensor signals and the program or control technique [1]. Important part is that this bachelor thesis deals with the simulation of field-level equipment, so the whole simulation consists of simulations of sensors and reactions of sensors on different input parameters.

Testing of the simulator will be executed with PLC on control level. So, the simulator will be connected to PLC and will behave exactly as a field-level system should.

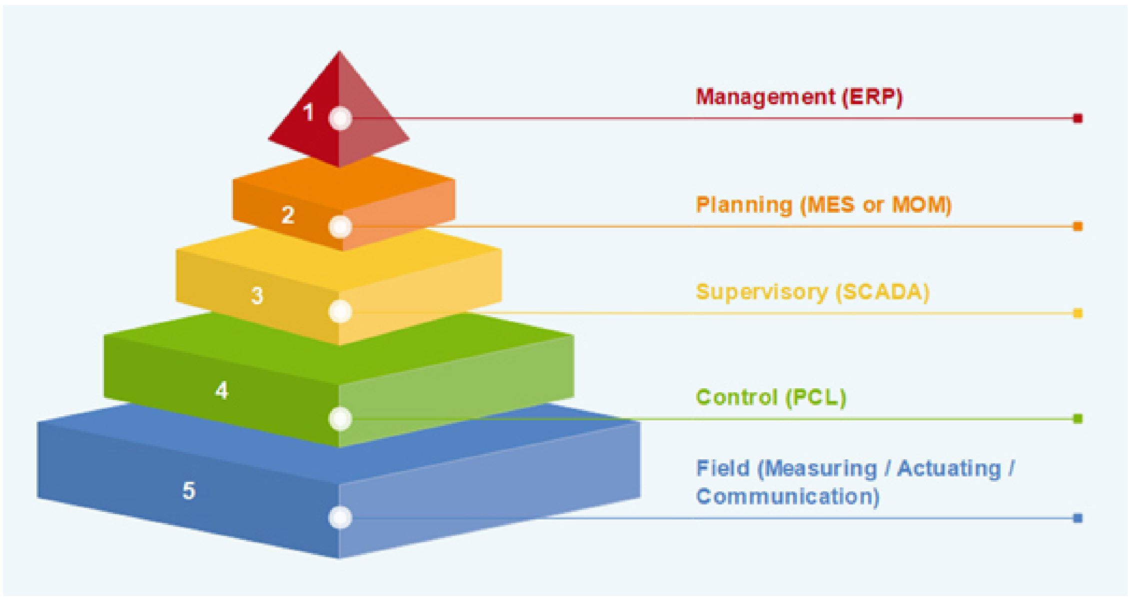


Figure 1.1: Pyramid of automation, [2019], In syspro.com, [Cit. 20.04.2020], [2]

1.1 Simulation of the field instrumentation

For a better understanding of the theme of the bachelor thesis, I need to define specifically what product system simulator is. It is a device, which uses the same amount of inputs/outputs as equipment, and it is programmed to behave the same way as simulated equipment i.e. it simulates the work of sensors and reactions of sensors on different input signals. The main demand for this kind of device is flexibility. Flexibility ensures that the device can be used to simulate different types of field-level equipment. Also due to the specific of the work of the simulator, that device will be doing, there are no industrial reliability requirements. So that's why it‘s preferable to choose Raspberry Pi instead of choosing the PLC.

The simulator is not only hardware but also a program, which is written in software. For this task three different platforms will be used: Python programming language with suitable libraries, Node-Red, which is a visual programming tool, and a developing environment, called “Mervis”.

The simulator can be also explained as a digital twin of a specific object for simulation. A digital twin is a digital replica of a physical entity. The digital representation provides both the elements and the dynamics of the simulating object [3]. All of this above applies to a definition of the simulator. So the simulator is a digital twin of a specific system, consisting of field level elements i.e. its digital copy.

So, to summarize the definition of the product simulator, it‘s a single-board computer Raspberry Pi with extension board UniPi attached and program developed in all the above-mentioned platforms i.e. it is a digital copy of the system reproduced with specific hardware (Raspberry Pi and UniPi).

Chapter 2

Specification of the system for simulation

This bachelor thesis deals with simulating the behavior of loaded linear pneumatic drive with monostable directing valve and bistable directing valve. But in the real world, there is a crazy variety of systems with different equipment. That equipment can be hydraulic, pneumatic, electrical equipment. And potentially users can add models of necessary equipment and use developed simulator as a template with basic systems included, such as linear pneumatic drive with monostable and bistable DCV, descriptions of which will be presented in the next chapter. So this bachelor thesis is the example of a digital twin of the specific system and can be used as the basis for making digital twin of system, with different equipment or working principle.

The task is to simulate two similar loaded field level systems. The difference between those systems is in the control valve. The first system is a pneumatic cylinder controlled by a mono-stable solenoid control valve. The second controls by the bi-stable solenoid control valve. In many applications it is required remotely control [4]. A mono-stable valve is held by spring into its home position. As soon as the valve actuates by a solenoid, the valve switches to its energized state. When the electrical power is released, the valve will return to its home position. These valves are also referred to as 'single-acting' solenoid valves. A bi-stable valve can be switched by a momentary operation and will remain in position. So, when the operation is stopped, the valve will not return to the initial situation before the operation. A bi-stable solenoid valve usually has a solenoid on both ends of the valve. Each solenoid is responsible for switching to a single state. These valves are called 'double-acting' solenoid valves [5]. Also, each system contents actuator. It will be the same actuator for both systems: double acting pneumatic actuator. “Double-acting” means that piston is used to extend and retract by pressure [4], not by spring for example.

It‘s required by the task of this bachelor thesis to design and test the way of simulation of selected one’s technological elements. To do this task I need to closely define the specific system.

2.0.1 Actuator with monostable valve

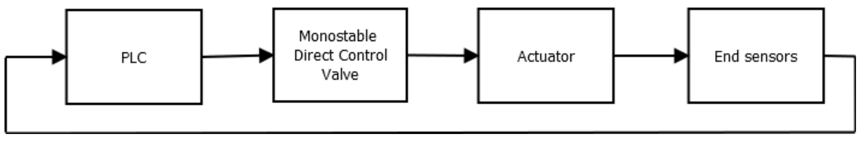


Figure 2.1: Block scheme for system with monostable valve

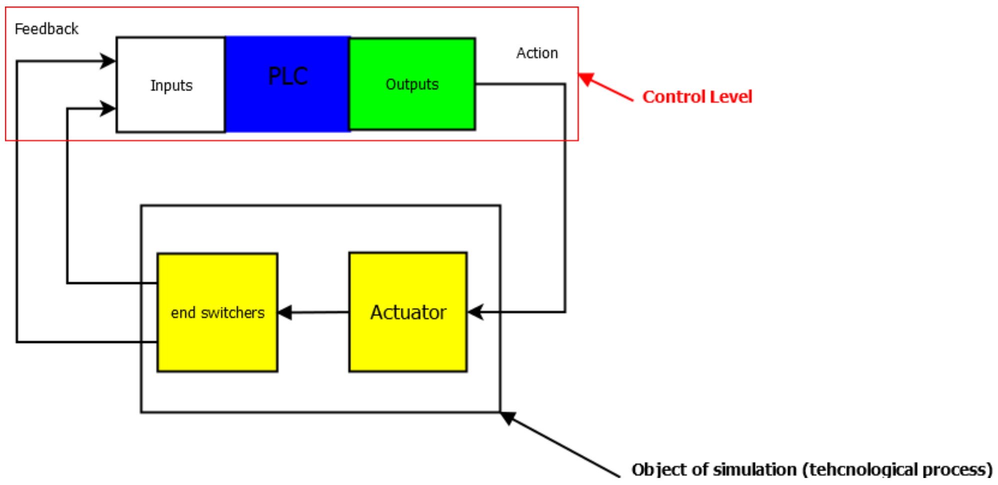


Figure 2.2: Scheme of the technological process for system with monostable valve

The part of the task is to make the simulation of this field-level equipment production unit, which is connected to the control level device - PLC. Direct Control Valve‘s solenoid requires external voltage powering 24 V and piston of the pneumatic cylinder is tracked by two end switchers therefore I have two inputs and one output connected, so I must simulate one input and two outputs on UniPi +RPi. Input on RPi+UniPi will be a simulation of the solenoid Y1 in Direct Control Valve. DCV is actuated by the solenoid and returned by spring. Outputs on RPi+UniPi will be the simulation of end switchers. On control level this production unit will be controlled by PLC

The system is functioning like this (the scheme is on 2.3): while PLC makes the output (Y1) True by directing the voltage on output, DCV (direct control valve), which is connected to the output, directs the air into the area beside the piston, it PLC makes the output False, DCV directs the air into the area in front of the piston. While the piston is in his starting position – the first end switcher (1M1) is triggered and DCV starts directing the air in the first area. When the piston is moving from starting position to end position – none of the switches are triggered, but DCV is still directing the air in the same area. When it reaches the end position – the second end switcher (1M2) is triggered and DCV starts direct air in the second area and that causes moving of the piston backward. After 1M2 switcher is no longer triggered, DCV should continue to move piston until it reaches 1M1, and start condition is created again. To explain the cycle better, there are block scheme (figure 2.1) and scheme of the technological process (figure 2.2) for cycle of the system with monostable valve.

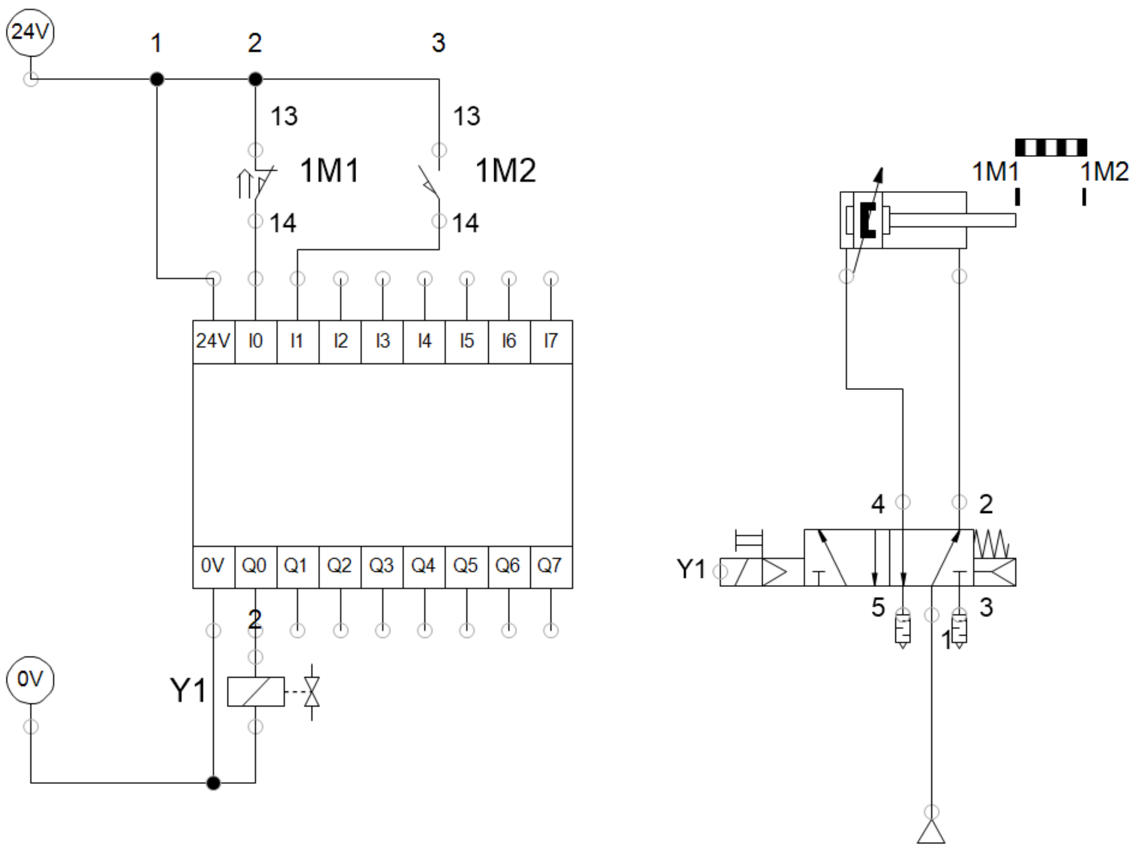


Figure 2.3: Scheme of the monostable task

2.0.2 Actuator with bistable valve

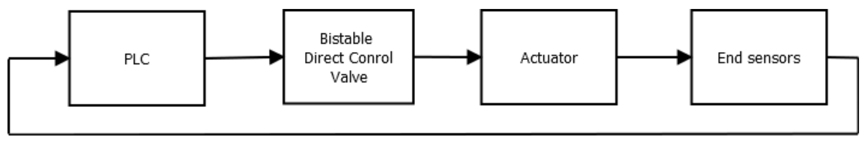


Figure 2.4: Block scheme for system with bistable valve

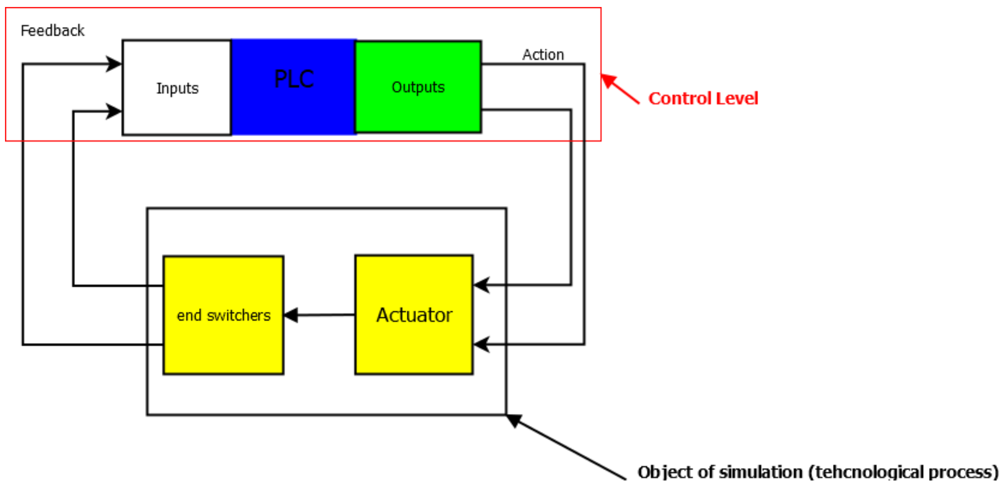


Figure 2.5: Scheme of the technological process for system with bistable valve

The other part of the task is to make the simulation of another field level system. This is the same production unit with a bi-stable control valve. As can be seen, the difference between systems is that the Direct control valve is actuated and returned only by solenoids therefore it has two stable positions. In that case, the valve is called bi-stable and requires two external voltage powering 24 V and, as in the previous system there will be two end switchers tracking the position of the piston, therefore there are two outputs from PLC and two inputs to PLC. It means that UniPi+RPi will require two inputs and two outputs for simulation. Two inputs on RPi+UniPi will be the simulation of the solenoids Y1 and Y2 in DCV. Two outputs on RPi+UniPi will be the simulation of end switchers.

The system is functioning like this (the scheme is on figure 2.6): Start state is when first end switcher (1M1) is triggered and Direct controlled Valve (DCV) is in his first position, directing the air in the area in front of the piston. After this PLC makes first output (Y1) True, DCV switches to the second position and starts directing air in the area beside piston, PLC makes first output False, none of the end switchers is triggered. When piston riches his end state, second end switcher (1M2) is triggered and PLC makes second output (Y2) True, this makes DCV to switch in his first position and to start directing the air in the area in front of the piston, which makes the piston to move back to his starting position, PLC makes second output (Y2) False.

The difference between the production unit with the mono-stable control valve and production unit with a bi-stable control valve is the number of stable positions. So, there is no need to put current through solenoids all the time, it would be enough to put current through on a limited amount of time, so DCV would switch, after this DCV would be in a stable position even after the current is removed from solenoid. It concerns both solenoids (Y1, Y2). To explain the cycle better, there are block scheme (figure 2.4) and scheme of the technological process (figure 2.5) for cycle of the system with bistable valve.

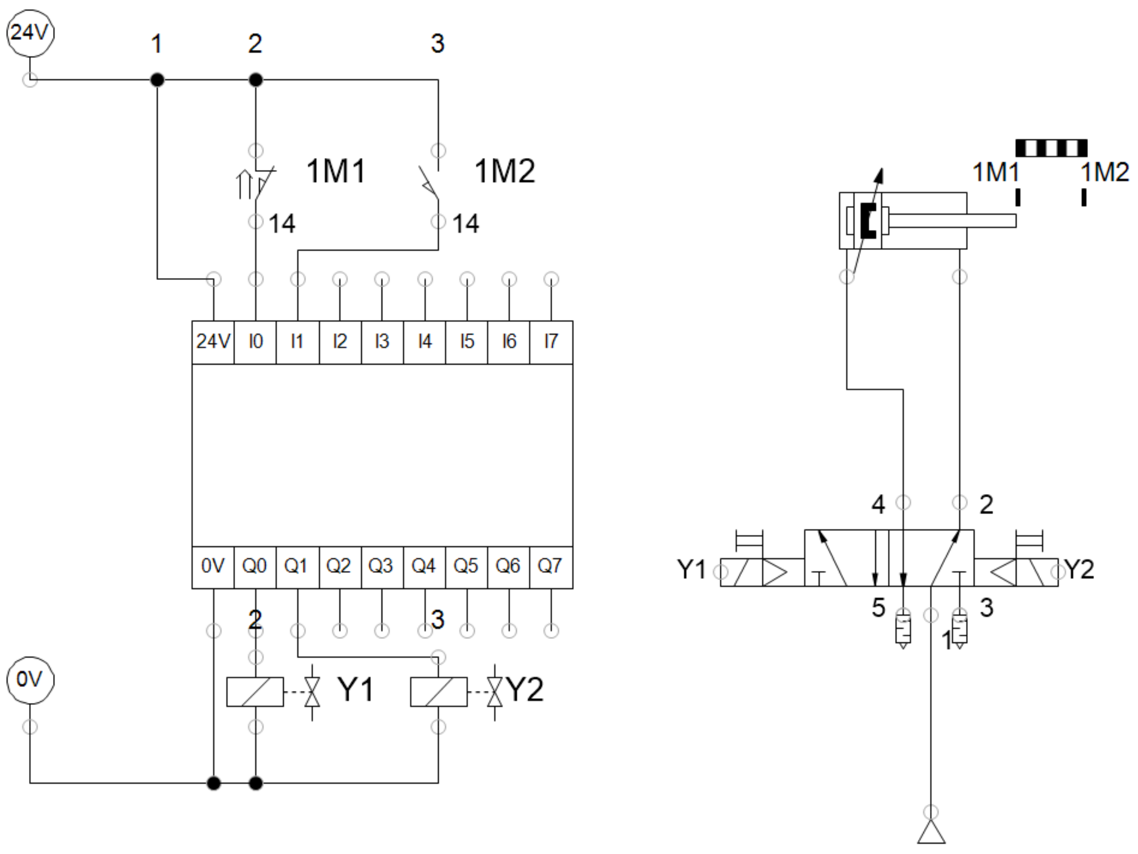


Figure 2.6: Scheme of the bistable task

Chapter 3

Raspberry Pi

Choice of Raspberry Pi as main peace of hardware was made, because of many reasons. Raspberry Pi is a relatively cheap and very widely used hardware. It has sizes of the credit card and still can be used as a personal computer, which means it is relatively powerful. RPi is provided with a UNIX based operating system, which means, that it has a lot of available platforms for developing and many ways to communicate (Graphical Interface, Command line, Connection through SSH). And most importantly, it was made to work with circuits as a controlling device.

Raspberry Pi is a series of small single boarded computers, developed in the United Kingdom. The model, which was used in this task is Raspberry Pi B+ v1.2. It‘s the 3rd Revision of Raspberry Pi. Its purpose was to promote and teach computer science and revive the microcomputer revolution of the 1980s. It was originally made for programming in Python language. The Raspberry Pi Foundation provides Raspbian, a Debian-based (32-bit) Linux distribution for download, as well as third-party Ubuntu, Windows 10 IoT Core, RISC OS, and specialized media center distributions. It promotes Python and Scratch as the main programming languages, with support for many other languages. The default firmware is closed source, while an unofficial open-source is available. Many other operating systems can also run on the Raspberry Pi. However, Raspbian is the recommended operating system for normal use on a Raspberry Pi. Raspbian is a free operating system based on Debian, optimized for the Raspberry Pi hardware. Raspbian is a Unix-like operational system. Most commands, which are applicable for Linux are applicable for Raspbian.

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards including B+ v1.2. Any of the GPIO pins can be designated (in software) as an input or output pin and used for a wide range of purposes [6].

Pins have two options: 3.3 V and 5 V. Out of 40 pins two 5V pins and two 3V3 pins are present on the board, as well as several ground pins (0V), which can't be changed. The remaining pins are all general-purpose, which have voltage 3.3 V [6]

Version, used for purposes of this bachelor thesis - Raspberry Pi B+ v1.2 series has powerful hardware for a credit card sized computer. Here are its specifications:

Figure 3.1: Raspberry Pi b+ v1.2

Official specifications:

CPU: quad-core ARM Cortex-A7, 1.2 GHz 64-bit.

Caches: 32kB Level 1 and 512kB Level 2 cache memory

System of chips: Broadcom BCM28367

Peripherals: GPIO 40 pins, HDMI, 3.5mm audio jack, 4X USB 2.0, Ethernet, Camera Serial Interface, Display Serial Interface.

The Raspberry Pi 3, with a quad-core ARM Cortex-A53 processor, is described as having ten times the performance of a Raspberry Pi 1. Benchmarks showed the Raspberry Pi 3 to be approximately 80 faster than the Raspberry Pi 2 in parallel tasks. On the Model B and B+, the Ethernet port is provided by a built-in USB Ethernet adapter. Although often pre-configured to operate as a headless computer, the Raspberry Pi may also optionally be operated with any generic USB computer keyboard and mouse. It may also be used with USB storage, USB to MIDI converters, and virtually any other device/component with USB capabilities, depending on the installed device drivers. None of the Raspberry Pi models have a built-in real-time clock. When booting, the time is set either manually or configured from a previously saved state at shutdown to provide relative consistency for the file system. The Network Time Protocol is used to update the system time when connected to a network [6].

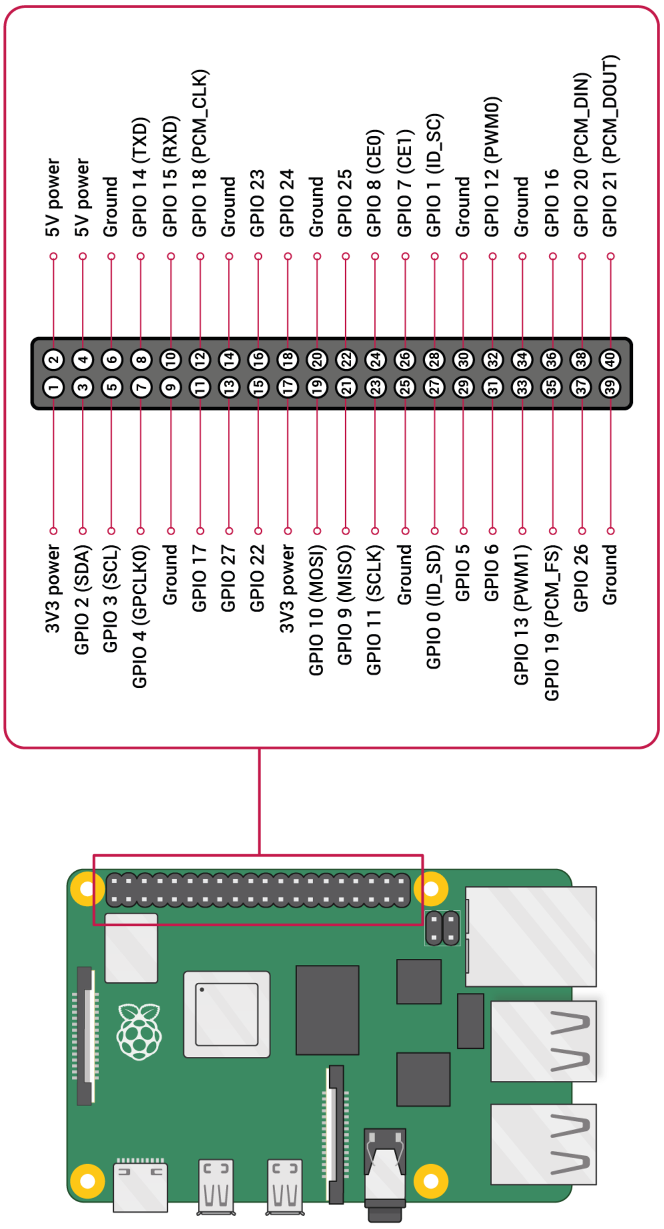


Figure 3.2: Raspberry Pi Foundation, GPIO port map, [online], [Cit. 13.04.2020] [7]

Chapter 4

UniPi

Unipi 1.1 is an extension board for Raspberry Pi computer which allows it to function as a programmable logic controller for monitoring, control, and regulation of automation systems. The board features a universal I/O architecture with digital inputs, a set of relays, analog I/Os, and broad connectivity thanks to 1-Wire bus, I2C, and UART ports[8].

UniPi v1.1 is the first model of extension boards from unipi technology. Despite this – the extension board allows Raspberry Pi to become fully functional PLC, only more flexible thanks to the bigger amount of available software and platforms to work with. The disadvantage of using RPi+UniPi as PLC is the absence of industrial qualities. But that‘s a perfect combination for product simulator.

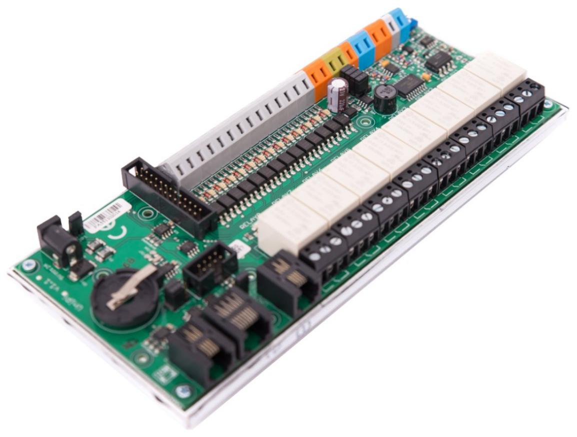
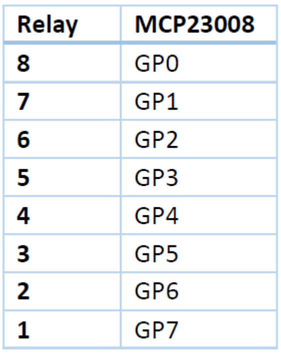


Figure 4.1: UniPi technology, UniPi v1.1, [online], [Cit. 14.04.2020],[8]

UniPi v1.1 has several building blocks. For this specific task, I will be using Relays and Digital inputs. Relays have some restrictions: maximum relay switching power is limited to 250V AC/5A or 24V DC/5A. I will be using 24 V external voltage, so it meets the requirements. Relays are controller by the MCP23008 (address 0x20), see map of MCP's GPIO to relays in table 4.1. MCP23008 is an expander, which allows RPi to control Relays building blocks on UniPi. Each relay has a LED indicating its state. There are three contacts for each relay named CO (change-over or C=connected), NO (normally opened), NC (normally closed). By default, contacts CO and NC are connected (NO is not connected), by switching the relay on, CO gets connected to NO (NC is disconnected)[9].



Relay	MCP23008
8	GP0
7	GP1
6	GP2
5	GP3
4	GP4
3	GP5
2	GP6
1	GP7

Table 4.1: UniPi technology, MCP23008 pin map, [online], [Cit. 20.03.2020],[9]

The other building block I will be using is Digital Inputs. These inputs can be triggered by 5-24V DC voltage with a minimum pulse length of 5ms. For easy visual reading of their states, all inputs are equipped with LED [9]. I used 24 V DC voltage from PLC, so this also meets the requirements. The problem appears to be that all inputs are primarily set to be driven by internal 12 V and in case of external power source the configuration of settings must be changed. The settings of external or internal voltage on digital inputs are regulated by jumper connectors JP2, JP3, JP4, JP5. In the following text, I will describe steps to configure jumpers for my exact task:

- JP2 is controlling the P02 port. When JP2 is switched on the side of the label – P02 acts like input for ground from an external power source. Otherwise, P02 is connected to internal 12 V. So, for my task I need JP2 to be switched to the side of the label.

- JP3 is controlling I01 and I02. When switched to the side of the JP3 label, inputs I01 and I02 act as inputs for signals from the connected peripheral device via the external power source. The ground of power supply for these inputs must be connected to the P02. So, for my task, I need JP3 also to be switched on the side of the label.

- JP4 is controlling I03 and I04 ports. When switched to the side of the JP4 label, inputs I03 and I04 act as inputs for signals from the connected peripherals via the external power source. The ground of power supply for these inputs must be connected to P01. For my specific task, I don't need to use I03 and I04 ports, so it doesn‘t matter if JP4 is switched to the side of the label or not.

- JP5 is controlling ports I05 – I14. When switched to the side of the JP5 label, inputs I05 - I14 act as inputs for signals from the connected peripherals via the external power source. The ground of power supply for these inputs must be connected to P01. For my specific task, I don't need to use I05 - I14 ports, so it doesn‘t matter if JP5 is switched to the side of the label or not.

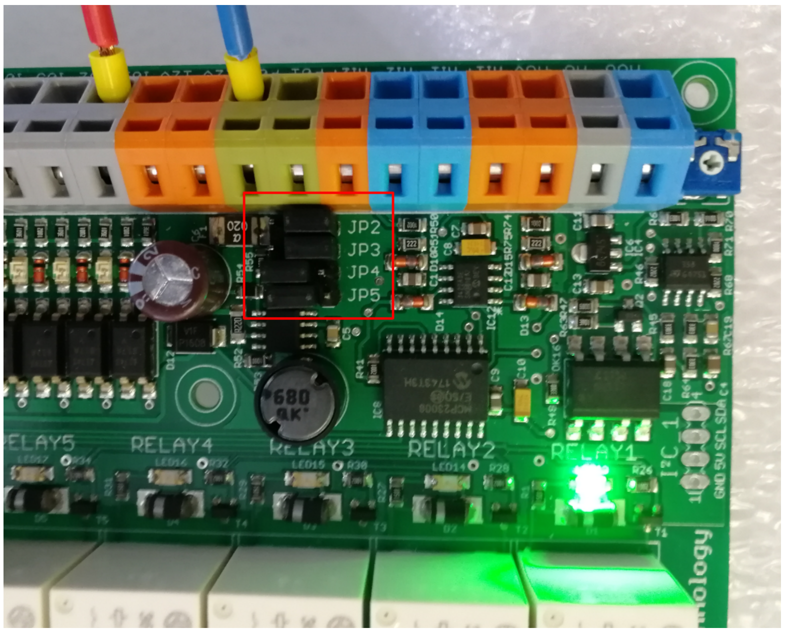


Figure 4.2: Configuration of JP2-JP5 connectors.

Digital inputs are connecting to Raspberry Pi via GPIO protocol. Before using this protocol, all connectors JP2-JP5 must be adapted to the purpose of the connection. After that, you can set up pins by addressing them with help of Raspberry Pi header map, see map in table 4.2.

UniPi	**RPi BCM**	**Function**	**Description**
AO	GPIO18	PWM	Analog Output 0-10 V
I01	GPIO04	Digital Input	Digital Input
I02	GPIO17	Digital Input	Digital Input
I03	GPIO27	Digital Input	Digital Input
I04	GPIO23	Digital Input	Digital Input
I05	GPIO22	Digital Input	Digital Input
I06	GPIO24	Digital Input	Digital Input
I07	GPIO11	Digital Input	Digital Input
I08	GPIO07	Digital Input	Digital Input
I09	GPIO08	Digital Input	Digital Input
I10	GPIO09	Digital Input	Digital Input

Table 4.2: UniPi technology, Raspberry Pi header map, [online], [Cit. 20.03.2020], [9]

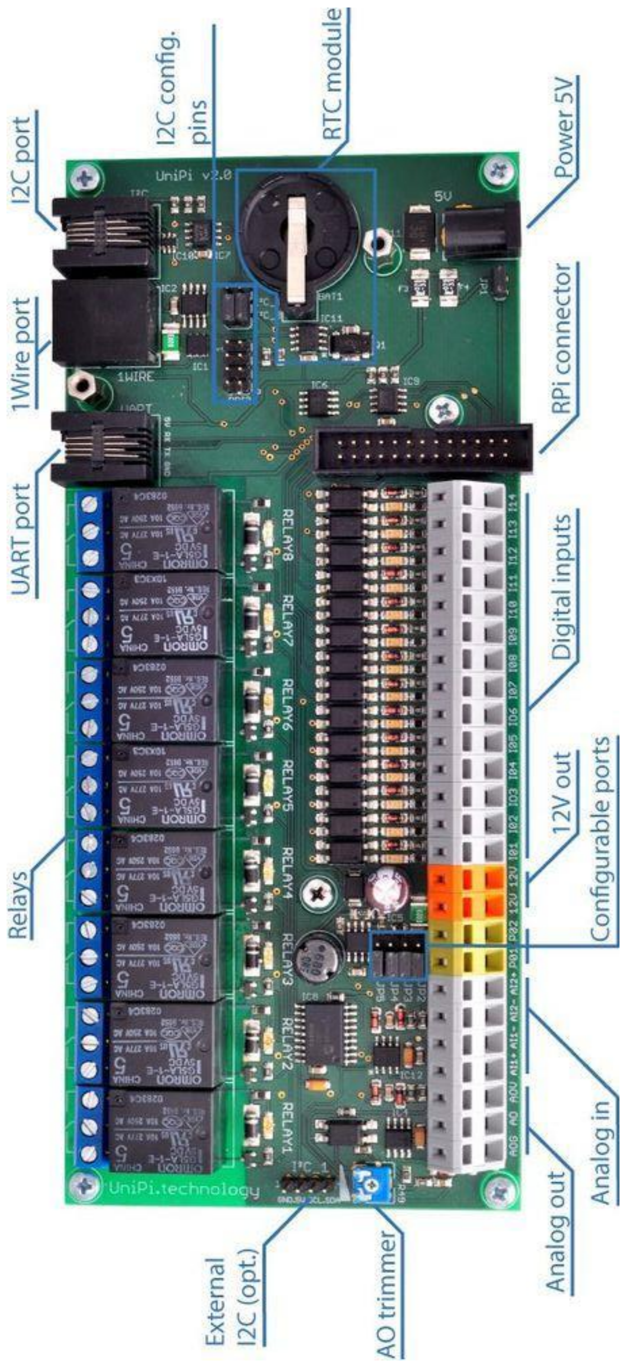


Figure 4.3: UniPi v1.1, peripheral map, Bc. Petr Simecek, Controlling system of smart house, (2017), [Cit. 17.06.2020] [10]

4.1 Communication between Raspberry Pi and UniPi

Because Raspberry Pi and UniPi are two separate boards, they must be connected to use UniPi as an extension board and to use its building blocks. UniPi has the slot for the flat connector from Raspberry Pi and Raspberry Pi is connected to UniPi v1.1 via GPIO connector. It connects GPIO inputs of UniPi v1.1 to RPi GPIO inputs.

Before connecting UniPi to RPi with GPIO connector, the option of powering of RPi+UniPi needs to be chosen. There are two options of powering: Single power source and Dual power source, and it depends on the JP1 connector. JP1 is a configurable connector, which is situated near input for a 5V power connector. If JP1 is switched to its label – Single power source mode is activated, which means – UniPi + RPi is powered through 5V DC 2A UniPi power connector. And if JP1 is switched against its label – Dual power mode is activated, which means that RPi is powered from micro USB port according to the RPi requirements and UniPi is powered from 5V DC 1.5A connector. After both boards are powered, the next step is to screw the spacers to the mounting holes of the UniPi, screw The Raspberry Pi to UniPi and connect provided flat cable.

As mentioned before, that flat cable connects Raspberry PI GPIO pins and UniPi GPIO pins, which are Digital inputs on the side of UniPi (grey inputs). So Raspberry Pi communicates to Digital Input building block though GPIO protocol. And Relays are connected to Raspberry Pi through MCP23008 expander. That‘s why RPi communicates with Relays via SMBus protocol. In the following text, I will explain how GPIO and SMBus protocol work.

4.1.1 GPIO

(general-purpose input/output) pins are uncommitted pins on a circuit board, whose behavior – including whether it acts as input or output is controllable by the user all the time. The main advantage of GPIO pins is that they have no predefined purpose. For using GPIO pins – the first thing to do is to define them as input or as output. Two 5V pins and two 3V3 pins are present on the board, as well as several ground pins (0V), which are unconfigurable. The remaining pins are all general-purpose 3V3 pins, meaning outputs are set to 3V3 and inputs are 3V3-tolerant. A GPIO pin designated as an output pin can be set to HIGH (3V3), logically "1", or LOW (0V), logically "0". A GPIO pin designated as an input pin can be read as HIGH (3V3) or LOW (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins, this can be configured in software [11].

GPIO is mainly purposed to communicate with other devices and extension boars such as UniPi. The GPIO peripheral has three dedicated interrupt lines. These lines are triggered by the setting of bits [12]. Those lines are: output connection, input connection, and enable line. Enable line controls if the pin is set to output mode or to input mode.

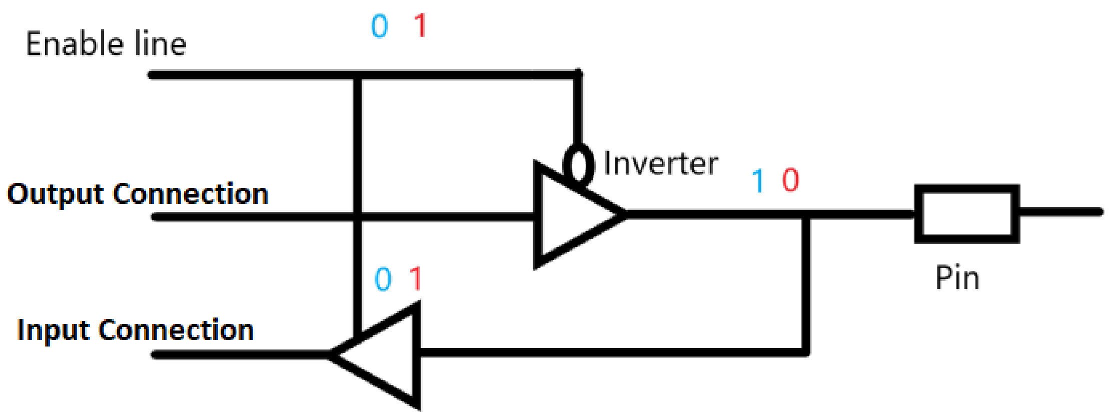


Figure 4.4: Scheme of GPIO pin

Blue numbers represent the state of logical "0" of enable line and red numbers represent the state of logical "1" of enable line. As can be seen in figure 4.4 – when the enable line is enabled, the output line gets activated, when enable line is disabled, the input connection gets activated. Triangles represent logical connections operators, however, enable line is connected to Output connection through logical NOT gate.

The Raspberry Pi GPIO interface is manufactured from CMOS components. CMOS is short for Complementary MOS, where MOS is a type of transistor. That means that NMOS and PMOS are complements of each other. NMOS transistors have their source connected to ground, whereas PMOS transistors connect their source to the positive supply [13]. Those CMOS components are implemented in output connection and input connection of GPIO pin.

Now I will describe separately output connection and input connection. Output connection of GPIO pin and Input connection of GPIO pin. More in figure 4.5 and figure 4.6.

In output connection (figure 4.5) red numbers represent states of elements if output connection is activated and HIGH or logically there is "1" written on GPIO pin, blue numbers represents the state of elements if the output is activated and LOW. If "1" is written on the input, then due to inverter logic MOSFET transistors are applied with LOW voltage. And when LOW voltage is applied to NMOS, it will not conduct. Vice versa, when LOW voltage is applied to PMOS, it will conduct. Due to the conduction of PMOS – Output connection will be pulled up to source and GPIO pin will be HIGH in output mode. Conversely, when Output connection is LOW – NMOS will conduct and

PMOS will not conduct. Due to the state of NMOS – The output connection will be pulled to the ground and GPIO pin will be LOW in output mode.

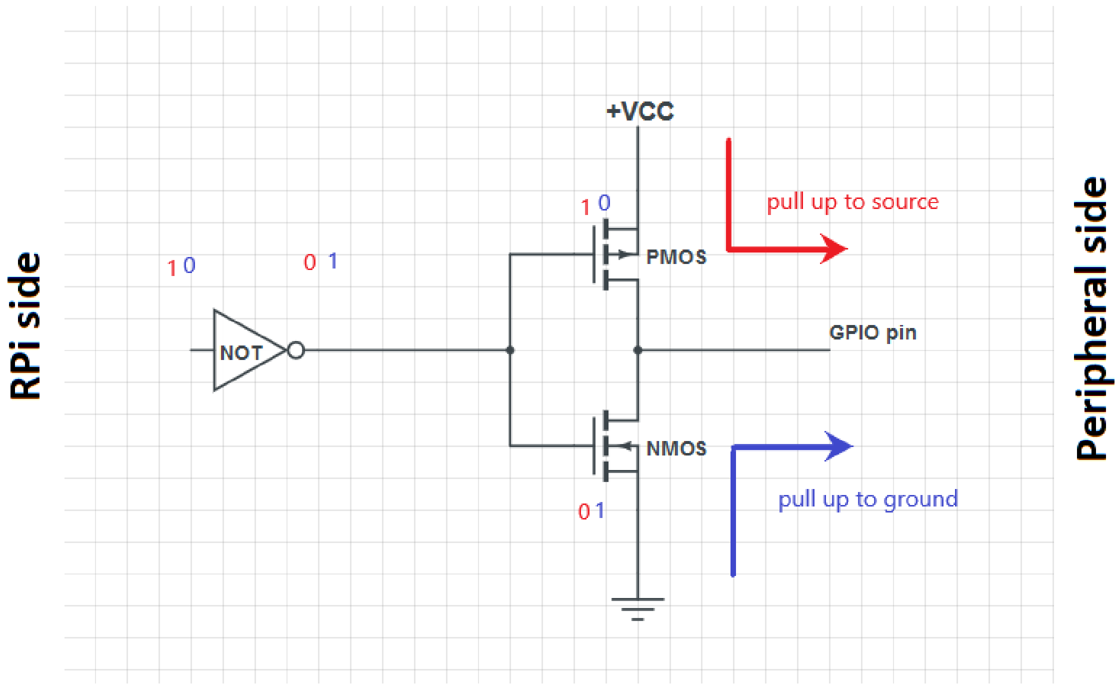


Figure 4.5: Scheme of output connection

Now follows a closer description of Input connection. More in figure 4.6. Red numbers represent the state, when GPIO pin is in input mode and HIGH, logically has “1”. Blue numbers represent the state, when GPIO pin is in input mode and LOW, logically got “0”. If GPIO pin is HIGH, due to inverter logic MOSFET transistors are applied with LOW voltage. And when LOW voltage is applied to NMOS, it will not conduct. And when LOW voltage is applied to PMOS, it will conduct. Due to the conduction of PMOS Input connection will be pulled to source and will have +VCC voltage, so logically “1” will be written on GPIO pin. Vice versa, when GPIO pin is in input mode and LOW, logically has “0”, due to inverter logic, the voltage on MOSFET transistors will be HIGH. When HIGH voltage is applied to the PMOS transistor, it will not conduct, bur when HIGH voltage is applied to the NMOS transistor, it will conduct. Due to the conduction of the NMOS transistor – Input connection will be pulled to the ground. This state means, that there would be written logical value “0” on GPIO pin.

Those were principles of functioning of GPIO pins. As well as simple input and output devices, the GPIO pins can be used with a variety of alternative functions, some are available on all pins, others on specific pins [11]. It can be used for pulse-width modulation and serial communication protocols, such as I2C and SPI. SPI interface is defined for usage through specific pins: GPIO19, GPIO21, GPIO18, GPIO17, GPIO16 pins. And I2C is defined for usage through specific pins: GPIO02, GPIO03, GPIO0, GPIO01 [11].

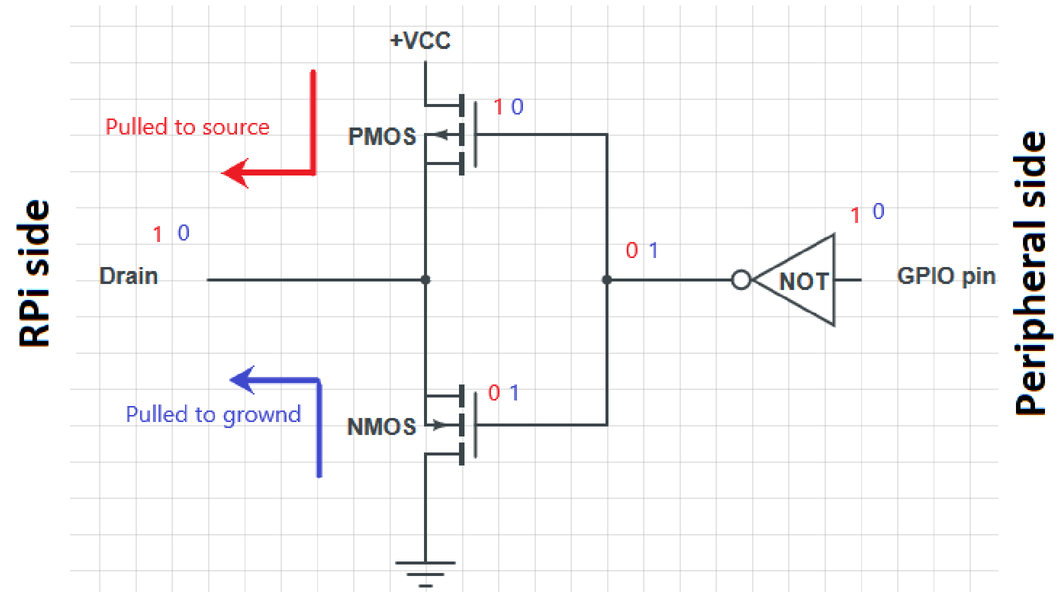


Figure 4.6: Scheme of input connection

In my task, I used directly GPIO pins to control Digital Inputs on the UniPi v1.1 extension board thought flat connector. Also, I used relays, which are controlled through I2C. Relays are controlled by MCP23008 expander, the expander is connected through the flat connector to Raspberry Pi‘s GPIO ports, which are specially defined for controlling and reading data with I2C interface (GPIO02, GPIO03, GPIO0, GPIO01). In the following text, I will closely describe the principles of I2C interface.

4.1.2 I2C

I2C (inter-integrated circuit) - synchronous serial, multi-master, multi-slave communication protocol. I2C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed, such as relays for expansion board. Unlike SPI, I2C uses only two wires for the entire process. These two wires are SDA (Serial Data) and SCL (Serial Clock). Typical voltages used are +5 V or +3.3 V, although systems with other voltages are permitted. I2C protocol can support multiple slave devices but unlike SPI, which only supports one master device, I2C can support multiple master devices as well. Every device sends/receives data using only one wire which is SDA. SCL maintains sync between devices through a common clock which is provided by the active master. Common I2C bus speeds are the 100 kbit/s standard mode and the 400 kbit/s Fast mode. There is also a 10 kbit/s low-speed mode [14].

I2C communicates with transactions. An I2C transaction consists of one or more messages. Each message begins with a start symbol, and the transaction ends with a stop symbol. Start symbols after the first, which begin a message but not a transaction, are referred to as repeated start symbols. Each message is a read or a write. A transaction consisting of a single message is called either a read or a write transaction. A transaction consisting of multiple messages is called a combined transaction. The most common form of communication is a written message providing intra-device address information, followed by a read message. Many I2C devices do not distinguish between a combined transaction and the same messages sent as separate transactions, but not all. The device ID protocol requires a single transaction; slaves are forbidden from responding if they observe a stop symbol. Configuration, calibration, or self-test modes that cause the slave to respond unusually are also often automatically terminated at the end of a transaction. More in figure 4.7.

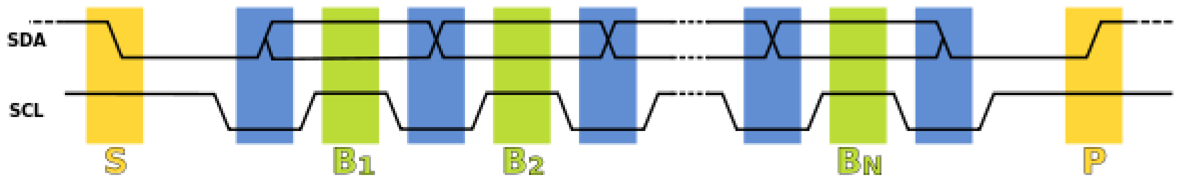


Figure 4.7: FLORYAN, Marcin. Data Transfer Sequence, [2017], In Wikipedia, [online], [Cit.20.03.2020], [14]

Sequence of actions during data transaction:

- J Data transfer is initiated with a start condition (S) signaled by SDA being pulled low while SCL stays HIGH.
- SCL is pulled low, and SDA sets the first data bit level while keeping SCL low (during blue bar time).
- The data are sampled (received) when SCL rises for the first bit (B1). For a bit to be valid, SDA must not change between a rising edge of SCL and the subsequent falling edge (the entire green bar time).
- The final bit is followed by a clock pulse, during which SDA is pulled low in preparation for the stop bit.
- A stop condition (P) is signaled when SCL rises, followed by SDA rising.

Chapter 5

Programmable Logic Controller

As mentioned earlier, the simulation will be tested with PLC (Programmable Logic Controller). PLC testing is necessary to make sure, that simulator will work correctly and mainly because the real system would be also connected and controlled by PLC. PLC program for testing the simulation must be identical to the program for controlling the real field level system.

IEC 61131 international standard defines requirements for modern controlling systems and PLCs. It has 5 basic parts: General information, Equipment requirements, and tests, Programming languages, User guidelines, Communication [15]. It doesn't depend on a certain firm or organization and has wild international support. For this chapter, IEC 61131-3 is more important, than other parts, because it defines programming languages. Programming languages on PLC can be divided into two groups: textual languages and graphical languages. Structured text (ST) and instruction list (IL) are textual languages and ladder diagram (LD), function block diagram (FBD) are graphical languages [16]. SFC is not official graphical language, but it comes from official languages and some PLC manufacturers are including it as an option for writing the program. The choice of the programming language for the project depends on the experience and preferences of the programmer. As a beginner in PLC programming I used Ladder diagram (LD) as it is the simplest language from all. This language comes from relay ladder logic and it serves for working with Boolean signals. A typical program, written in LD consists of typical elements: Power rail (vertical line on the left), Neural rail (vertical line on the right), and elements between them, which re-exposed to a current going through them (more on figure 5.1). In reality, the ladder logic diagram is only a symbolic representation of the computer program. So, power does not really flow through any actual contacts.

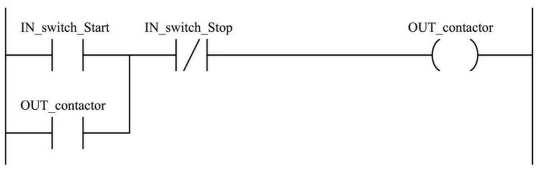


Figure 5.1: Simple program in LD

Elements in LD are limited to few options. Normally open contact (figure 5.2(a)), normally closed contact (figure 5.2(b)), coil (figure 5.2(c)), negated coil (figure 5.2(d)), set coil (figure 5.2(e)) and reset coil (figure 5.2(f)). Basically all common tasks for PLC can be implemented with those elements. But harder tasks require more sophisticated and complex programs, therefore require other programming languages.

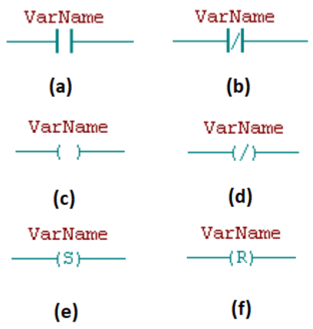


Figure 5.2: Basic elements LD

Now I will closely describe, what is PLC. Programmable Logic Controllers are at the forefront of manufacturing automation. Many factories use Programmable Logic Controllers to cut production costs and/or increase quality. PLCs developed out of the need to replace the hard-wired relay panels. In the 1960s, a typical automated assembly or other manufacturing line had a cabinet of relays wired to control the operation. As one might expect, debugging relay failures could be time-consuming, and changing functionality by modifying the sequence of operations was time-consuming and costly because of the required rewiring [17].

The architecture of a general PLC is shown in figure 5.3. The main parts of a PLC are its processor, power supply, and input/output (I/O) modules. In a micro PLC, all three main parts are enclosed in a single unit. For larger PLCs, these three parts are separately purchased [17]. The architecture of the PLC is the same as a general-purpose computer. In fact, some of the early PLCs were computers with special I/O. However, some important characteristics distinguish PLCs from general-purpose computers. They can be placed in an industrial environment that has extreme temperatures (typically up to 70 C), high humidity (up to 95 percent), electrical noise, electromagnetic interference, and mechanical vibration. They are easy to use by plant technicians. Hardware interfaces are easily connected. Modular and self-diagnosing interface circuits pinpoint malfunctions and are easily replaced [17].

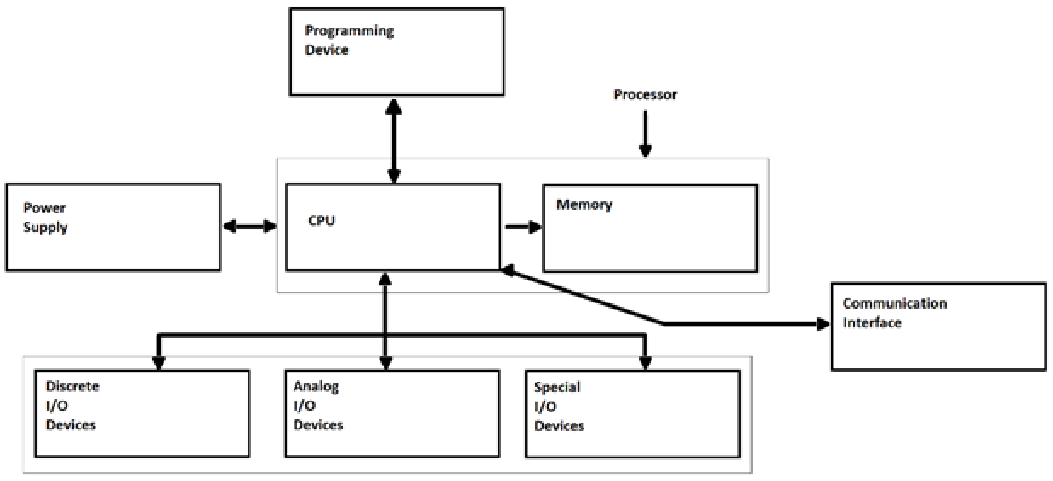


Figure 5.3: ERICKSON, Kelvin. Architecture of typical PLC, 1996, In IEEE potencials, [Cit.15.04.2020], ISSN 0278-6648, [17]

Now in the following text, there will be an explanation: how PLC executes the program with ladder logic. PLC works in cycles, so it repeatedly doing the programmed cycle. During the cycle, the PLC repeatedly executes a scan, during which the input channels from all of the input modules are copied into the internal memory; the ladder logic is scanned, updating the outputs being held in internal memory, and then the internal outputs are copied to the actual output modules. After the actual outputs have been updated, the scan is repeated. The time to execute a scan, depending on the number of I/O channels and the length of the ladder logic program, is on the order of 1 - 10 milliseconds. Normally, the processor uses only the internal copy of I/O when executing the ladder logic. It does not read input channels or write output channels. However, some manufacturers do allow that option, which is useful in critical or emergencies [17].

In conclusion, it can be said, that Programmable Logic Controllers are the main equipment for factory automatization. It uses unique ladder logic for programming. Ladder logic is implemented with three elements: normally opened contact, normally closed contact, and output. The working principle of PLC is cycle working. The cycle consists of checking inputs, executing programs, and updating the outputs, after that cycle repeats.

Chapter 6

Conclusion of Theoretical part

The theoretical part of this bachelor thesis serves a purpose to be explanation and preparation for the practical part. So in this part of the bachelor thesis, I've explained important working principles of some mechanisms that are important in making a simulator with Raspberry Pi computer and added expansion board.

In chapter 1 was presented the pyramid of automation and defined, which levels of the pyramid are involved in this bachelor thesis. Also, the definition of a product system simulator was given.

In chapter 2 were defined two systems for simulations: mono-stable and bi-stable, were shown their differences, which include different Control Valves and a different program for Control Level equipment because of this difference. Introduced the pneumatic schemes of those systems and presented the block scheme and scheme of the technological process for each system. Defined equipment, which is used to create this type of systems in real life, described it, it‘s behavior, and a sequence of actions in the working cycle of the systems. Also, it was said, that those systems are working with a load.

In chapter 3 hardware for simulator stated to define. I had started with Raspberry Pi. Its official specification was cited, and defined specific equipment of RPi, I will be using it in the practical part. This specific equipment is GPIO pins.

In chapter 4 defining the hardware with a description of expansion board for RPi - UniPi v1.1 was continued. It‘s equipment, used for the practical part, was described and some pin maps, was cited. Also, the methods of communication between RPi and UniPi was showed and described in detail: GPIO pins of UniPi are connected directly to GPIO pins of RPi and relays are controlled through MCP23008 port expander with I2C serial protocol.

And chapter 5 finishes with a description of control level equipment, used for testing of my simulator and most importantly used to control this type of the system in an actual production line and actual manufacture - PLC. The brief description of IEC 1131 standard and IEC 1131-3 part was given, specifically one with ladder logic and it's symbols.

Chapter 7

Physical connection of UniPi and PLC

At the beginning of the practical part, I want to start with an explanation of the physical connection between UniPi and PLC. I described it in 1.2.1 and 1.2.2 chapters, but here I will describe how PLC is connected in detail to the hardware.

7.1 Connection for simulating the system with monostable valve

I will describe connections from both sides. From the side of UniPi and the side of PLC. I'll start with UniPi. The input from PLC is connected to I01 and port P02 serving as a common ground. Relay outputs RELAY1 and RELAY2 must be powered by an external 24V supply.

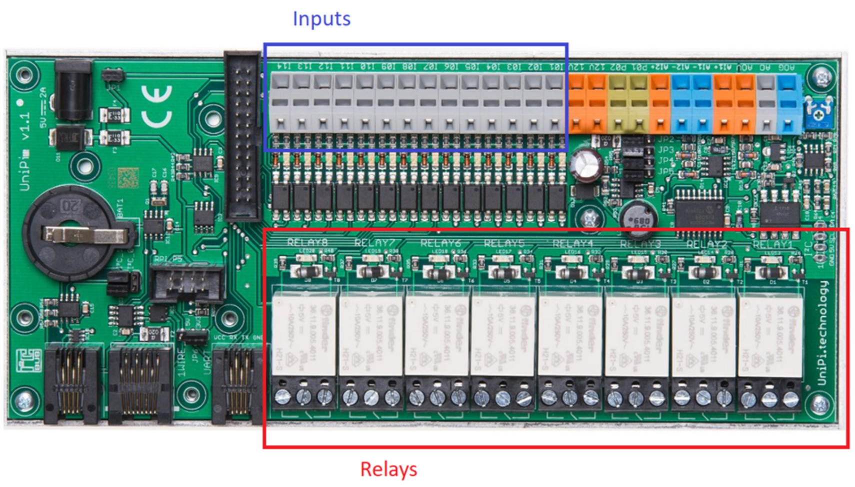


Figure 7.1: Scheme of UniPi

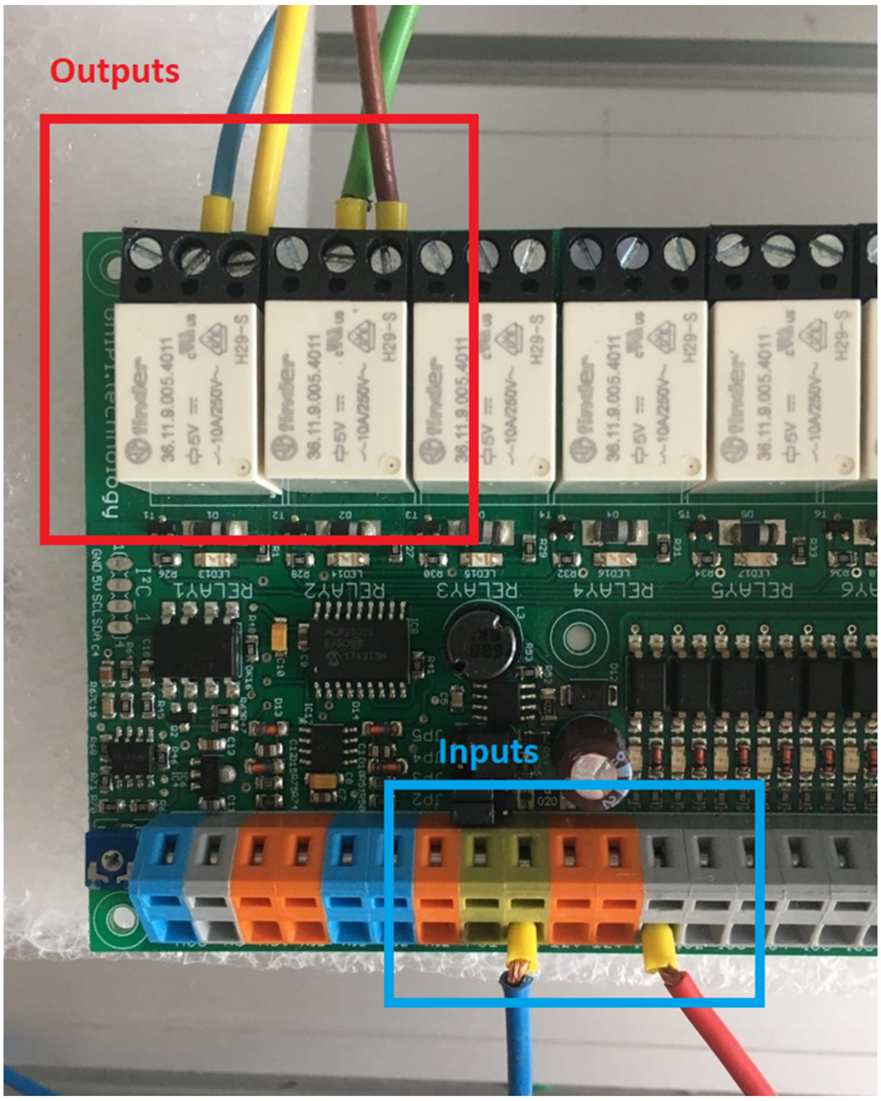


Figure 7.2: Actual connection of UniPi

Yellow and brown output wires, which are connected to RELAY1 and RELAY2 are external 24 V powering for relays to simulate end switchers. Those wires are connected to normally open contact. Armatures of the relays (blue and green output wires) are connected to the inputs of PLC. So, when relays are switched, 24 V voltage is directed to inputs of PLC. Blue input wire, which is connected to P02 input (green port) is also connected to the GND because with the current position of configurable connectors JP2-JP5 – P02 acts as GND for I01 and I02 ports (more in chapter 3). Red input wire which is connected to I01 (grey port) is also connected to PLC, but to output.

Now with the connection from the side of the PLC. There are two inputs to PLC and one output. Two inputs are connected to RELAYS and the output from PLC is connected to the I01 (grey port at figure 7.2). The PLC is situated in the stand and has all inputs and outputs connected to the front desc of the stand. It‘s necessary to:

Sequence of actions of connecting PLC to UniPi

- Connect middle contact (armature) of Relay 1 to digital input 0 of the PLC (RELAY1 to DI0)
- Connect middle contact (armature) of Relay 2 to digital input 1 of the PLC (RELAY2 to DI1)
- Connect normally open contacts of Relays to 24 V external powering
- Connect digital input 1 of the board to digital output 0 of the PLC (I01 to DQ0)
- Connect protective output 2 to ground (PO2 to GND)

The whole picture of the connection between UniPi and PLC can be seen in the figure 7.3.

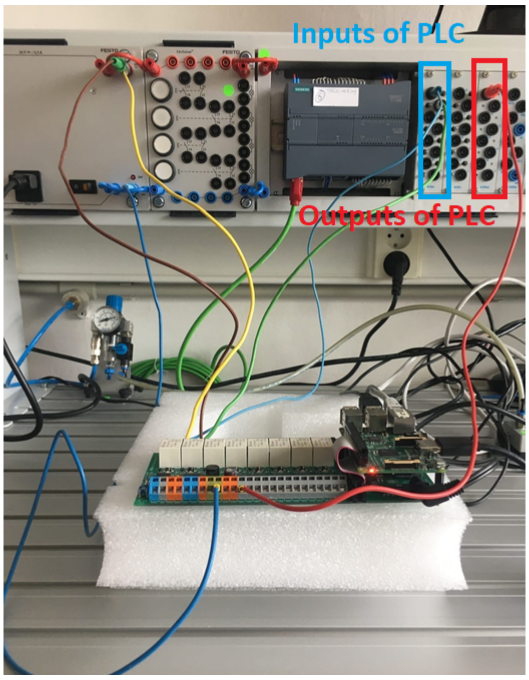


Figure 7.3: The whole picture of the connection between UniPi and PLC

Circuit scheme of connection between PLC and UniPi v1.1 for simulating the system with a monostable valve can be seen in figure 7.4.

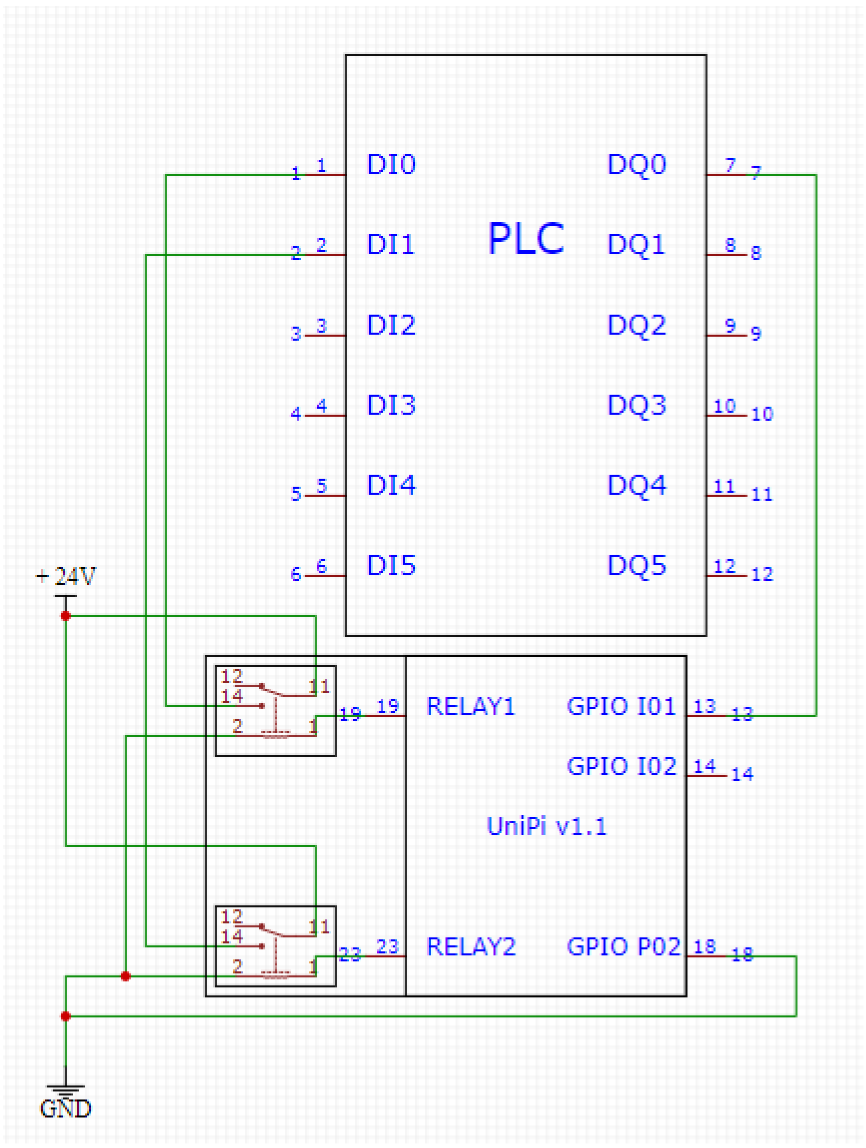


Figure 7.4: Circuit scheme for simulating system with monostable valve

7.2 Connection for simulating the system with bistable valve

The system with a bistable valve has one additional input to the UniPi because bistable DCV has two solenoids, therefore I need to simulate two solenoids, therefore I need two inputs. So I will start describing connections from the side of UniPi, I02 and I01 are serving as input ports from PLC, P02 acts as the ground for them due to the configuration of JP connectors (more in chapter 3). Relays are connected to the 24 V external power source through the normally open contact and armature of Relay 1 and Relay 2 is connected to the PLC. You can see the connection on the figure 7.5.

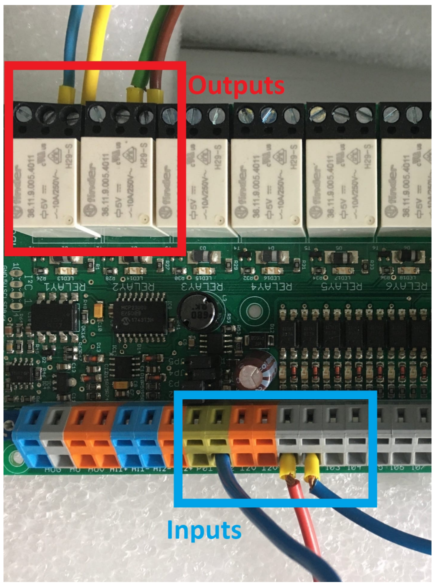


Figure 7.5: Actual connection of UniPi, bistable valve

Now connection from the side of the PLC. I01 and I02 are connected DQ0 and DQ1 outputs, normally open contacts of Relays are connected to DI0 and DI1 inputs. Actual picture is on the figure 7.6.

Sequence of actions of connecting PLC to UniPi

- Connect middle contact (armature) of Relay 1 to digital input 0 of the PLC (RELAY1 to DI0)
- Connect middle contact (armature) of Relay 2 to digital input 1 of the PLC (RELAY2 to DI1)
- Connect normally open contacts of Relays to 24 V external powering
- Connect digital input 1 of the board to digital output 0 of the PLC (I01 to DQ0)

- Connect digital input 2 of the board to digital output 1 of the PLC
  (I02 to DQ1)

- Connect protective output 2 to ground
  (PO2 to GND)

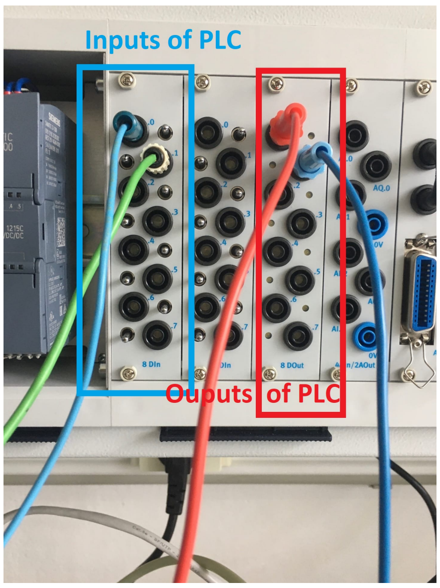


Figure 7.6: The whole picture of the connection between UniPi and PLC, bistable valve

Circuit scheme of connection between PLC and UniPi v1.1 for simulating the system with a bistable valve can be seen in figure 7.7. It doesn't differ much from connection for simulating the system with a monostable valve, the only difference is the connection between DQ1 and GPIO I02.

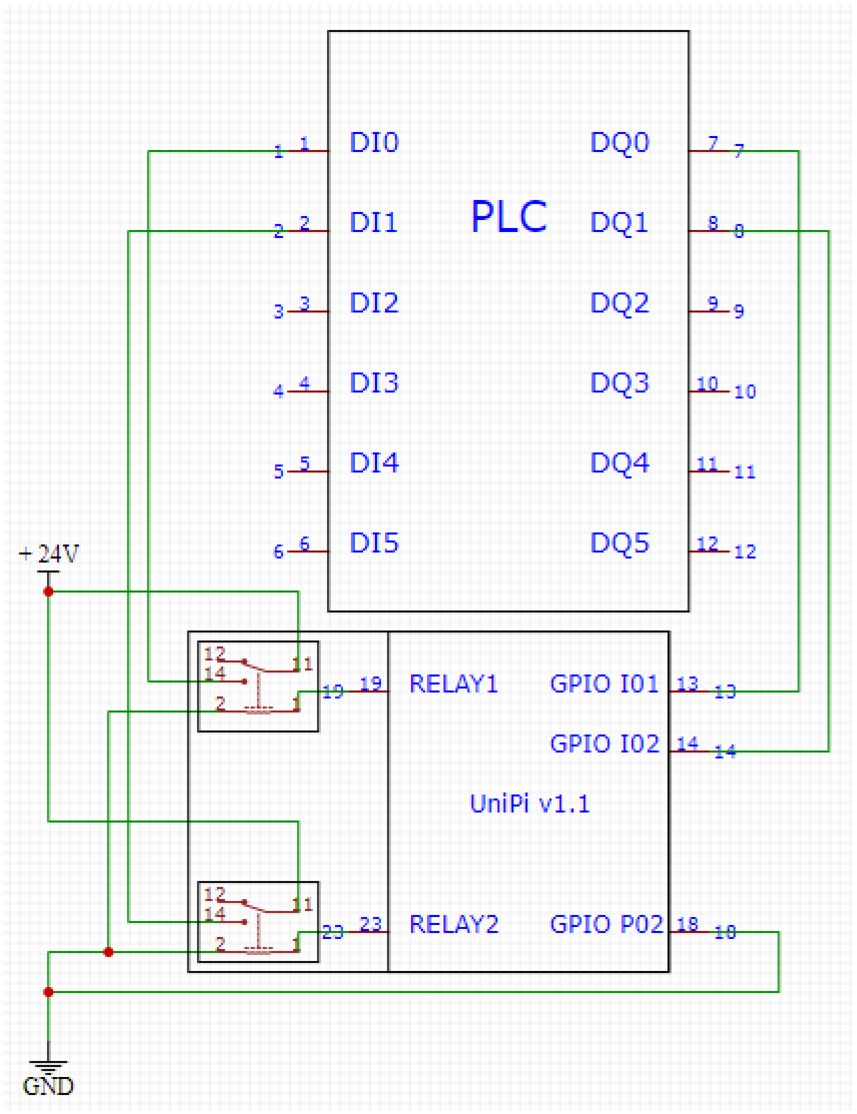


Figure 7.7: Circuit scheme for simulating system with bistable valve

Chapter 8

PLC program

I think I should begin practical part of this bachelor thesis with a description of created PLC programs, because PLC program is common for real field level system and simulator. So in this chapter I will describe in detail the created program for PLC and how it works with the real system. It should also give basic principles for a simulator. The first part will be about the work of PLC program for the system with mono-stable DCV, second part will be about the work of PLC program for the system with bi-stable DCV. All programs are written in ladder logic, which is the first one of the program languages for PLC and remain dominating for its simplicity in understanding. First program is written in commercial software TIA PORTAL v15 [18] (Totally Integrated Automation Portal), which is intelligent development environment (IDE) designed by Siemens. The software is compatible with siemens designed PLC‘s and supports ST, FBD and LD languages. Second program is written in PLCopen editor[19], due to COVID 19 quarantine. Because of lack of the access to physical PLC I needed to use PLCopen project. OpenPLC is an open-source Programmable Logic Controller that is based on an easy to use the software. The OpenPLC project was created under the IEC 61131-3 standard, which defines the basic software architecture and programming languages for PLCs [20]. The later program was tested on the TIA PORTAL with real PLC and worked properly.

8.1 PLC program for monostable system

- DI0 - Digital Input 0 (on figure 8.1), first end switcher 1M1 (on figure 2.3)
- DI1 - Digital Input 1 (on figure 8.1), second end switcher 1M2 (on figure 2.3)
- DI0 - Digital Output 0 (on figure 8.1), coil of relay Y1 (on figure 2.3)

As it is possible to see in figure 8.1, the program I've created is extremely simple. It does the function of a light switcher with two buttons. While input DI0 is True AND input DI1 is False - output DQ0 becomes True and starts to support condition of himself being HIGH but as long as DI1 stays False. When DI1 becomes True condition breaks and DQ0 becomes False.

Digital Inputs 0 and 1 simulate end switchers of the system (1M1, 1M2 on figure 2.3), because they are connected to RELAYS 1 and 2 on UniPi board and relays are powered 24 V from an external source. So when relays are switched PLC gets similar 24 V signal on inputs as it would get from end switchers. Strictly speaking, relays are switchers itself. DQ0 simulates the solenoid in DCV (Y1 on figure 2.3), so when this OUTPUT is True - DCV is actuated and directs air in the area beside piston to move it. If the output is False - DCV is spring-returned to its initial condition and directs air in front of the piston.

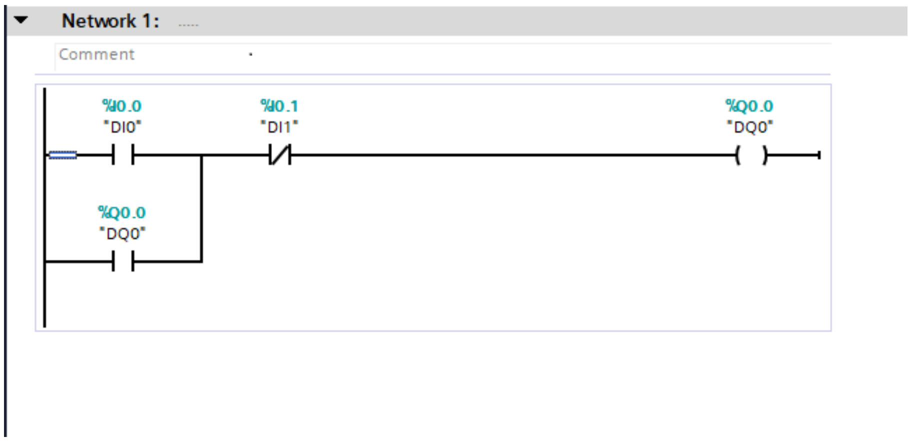


Figure 8.1: PLC program for mono-stable task, written in TIA Portal v.15 [18]

So in the first state of the real mono-stable system when 1M1 switcher is on and 1M2 switcher is off (more in figure 2.3) - Y1 solenoid should be True because DCV needs to be actuated to direct air beside the piston. That corresponds to the program, while DI0 is True and DI1 is False - DQ0 is True. In the second state - transition state, Y1 still needs to be actuated to move the piston to the third "end" state, although none of the end switchers are pressed. The program does the same, while DQ0 is True, it will remain True from the previous state, AND DI1 (1M2 switcher) is False - DQ0 remains True. And in the third state 1M2 switcher is pressed (DI1 becomes True), so condition for DQ0 to stay True is broken, therefore Y1 (DQ0 on figure 8.1) is False and DCV is spring returned, therefore air is directed in front of the piston until it will press 1M1, DI0 becomes True again and it will create the condition for the first state.

8.2 PLC program for bistable system

- M1 - Digital Input 0 (on figure 8.2), first end switcher 1M1 (on figure 2.6)

- M2 - Digital Input 1 (on figure 8.2), second end switcher 1M2 (on figure 2.6)
- Y1 - Digital Output 0 (on figure 8.2), coil of the first relay Y1 (on figure 2.6)
- Y2 - Digital Output 1 (on figure 8.2), coil of the first relay Y2 (on figure 2.6)

The program for the bi-stable system is even simpler than the mono-stable program. The differences between them are the method of controlling the DCV and the number of outputs. Bi-stable DCV is actuated and returned by solenoids, so two outputs are needed to be controlled. Also for actuating or returning DCV, the current must go only through one solenoid for a short amount of time. Through first solenoid (Y1 on figure 2.6 and on figure 8.2) - for DCV to actuate, through second solenoid (Y2 on figure 2.6 and on figure 8.2) for DCV to return. In stable positions, both solenoids are off. M1 and M2 represent end switchers 1M1 and 1M2 on figure 2.6.

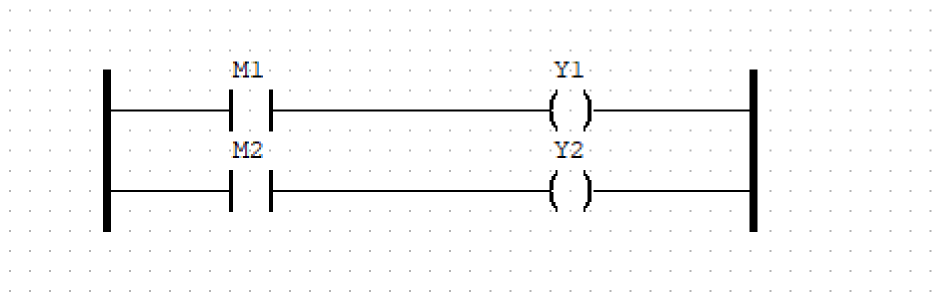


Figure 8.2: PLC program for bi-stable task, written in PLCopen [19]

In the first state of real bi-stable field level system first end switcher (M1 on figure 8.2) is pressed and DCV needs to be actuated to move the piston to its end position, so Y1 needs to be True. That responds to the program as long as M1 is pressed Y1 is True. In the second state, the transition is happening, so no switchers are pressed and no current goes through solenoids because DCV is in a stable position and continues to move the piston. It happens until the second end switcher is pressed and DCV needs to be returned in his original stable position to move piston backward. The program does the same. When second switcher (M2 on figure 8.2) is pressed, so M2 is True, Y2, which represents second solenoid (more on figure 2.6) becomes True too.

The only problem, that could arise, is that M1, M2 are not being held True long enough, so Y1 and Y2 would not be True long enough too. Therefore solenoid would not be able to push DCV in his other stable position. But in real life end switchers have a width so M1 and M2 are being held for some amount of time. So I reflected it in simulator too by adding some delay for RELAYS as the reaction time of sensors.

Chapter 9

Development of the program on UniPi+RPi

The simulator was made on three different platforms. In this chapter development of the simulator will be described in detail on each platform. Platforms are Python, Node-RED, and Mervis. Python is a widely used program language with modules, that can be suitable for many situations and usages. That makes python a very flexible choice for development visualization and monitoring graphs for the simulator, but sometimes over-complicated for some purposes, like communicating with relays and pins. That can be realized with much simpler instruments. Node-Red is simpler in that way because it‘s a mostly visual programming and it can be also widely implemented. It can be started on PC, Raspberry pi, and controlled with another PC or Raspberry Pi through a local network. Mervis is also relatively easy for using, because of the opportunity of visual programming, but it can be used only with UniPi technology products, so it‘s not as widely used as Node-Red and Python.

As mentioned before, Python is a widely implemented program language with a big amount of modules for a wide range of purposes. Python is a standard built-in language for Raspberry Pi, so it‘s logical to use Python as one of the platforms. Python v.3.7.3 had been used for making this simulator. It‘s one of the latest versions, the latest one at the moment is Python v. 3.8.2.

Node-RED [21] is flow-based visual development tool. Node-RED provides a web browser-based flow editor, which can be used to create JavaScript functions. Elements of applications can be saved or shared for re-use. The runtime is built on Node.js., which is the environment for running java code outside the browser. Developed originally by IBM for wiring together hardware devices [22]. I used Node-RED v.1.0.3 for the simulator, the latest version at the moment is v.1.0.6. Node-RED v.1.0.3 has problems in compatibility with Raspberry Pi v.1.2, sometimes RELAY wasn‘t reacting to commands as it should, so

I assume the problem would be in I2C protocol, maybe they are using a different version of it. Other than that Node-RED is a very interesting platform with simple programming with lots of predefined functions.

Mervis [23] is an intelligent developing environment (IDE) designed by UniPi technology especially for UniPi controllers and extension boards, that‘s why it was chosen as the third platform. A development environment for creating, debugging, and remote management of control programs for Unipi controllers. The Mervis IDE supports two IEC 61131-3 compliant programming methods for creating controller behaviour: FBD, ST [24]. For its usage necessary to install on RaspberryPi another version of Raspbian especially changed for better communication with Mervis IDE.

9.1 Development in Python

I had written two simulation programs in Python. One for the mono-stable system, one for the bi-stable system. But differences between them are minimal. So at first, I will consider the program for a mono-stable system and then describe differences of the program for the bi-stable system over the mono-stable system. The Code of the simulator consists of the 3 main parts: import of the modules, written functions, and main code.

9.1.1 Import of modules

The first part is the import of modules, which I need for the program:

```
import time
import threading
import sys
import copy
import pygame
import math
from matplotlib import pyplot as plt

try:
    import smbus
    import RPi.GPIO as GPIO
    DEBUG = False
except:
    DEBUG = True
```

"Time" module is used in measuring how long one cycle has to repeat and to delaying the cycle at endpoints, because of the reaction of the sensors. "Threading" module used for running two functions in parallel, because in the simulator I need to run the visualization part and simulation part simultaneously. "Sys" is imported only to use one function from it:

```
sys.exit()
```

It is a function, that helps a developer to exit the python. It is used in the visualization part for when the user is clicking on the "X" sign of the window - visualization will stop working. The "Copy" module helps to copy variables. "Pygame" is the core of the visualization part of this program. It‘s a set of modules, that allows to draw different forms and show them on a separate window. "Math" is a module, that allows to run some mathematical operations. "SMBus" is a module, that helps to communicate with building blocks of the UniPi. It communicates with I2C serial protocol. It‘s named after system management bus protocol, which is the same as I2C protocol but uses 3.3 voltage instead of 5V. "GPIO" is module for communicating with GPIO ports of RPi. And "pyplot" submodule from "matplotlib" is the perfect tool for drawing the plot, it was used to draw the plots in real-time. The import of the last two modules is conducted only on UNIX systems, because of their incompatibility with windows. That "try, except" construction in the end, is importing those modules, only if code is running without errors.

9.1.2 Simulation function

The second part of the code is programming the functions, which I will be using in the main code part. **The first function is simulation part**. It‘s the function with two arguments which returns no calculations, no variables on the end of it, but runs repeatedly and executes the commands, such as: "switch the first relay and after the defined time of the cycle switch the second relay".

```
# RPi implementation:
def simulation_part(lock, payload):
    t = tc

    # setup GPIO
    GPIO.setmode(GPIO.BCM)
    channel = 4
    GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)

    # setup I2C relays
    bus = smbus.SMBus(1)

    bus.write_byte_data(0x20, 0x00, 0x00)
    bus.write_byte_data(0x20, 0x09, 128)
    # setup real time counter
    stime = time.time()
    ref = time.time()
    a = time.time()
    b = time.time()
    while (b - a) < r:
        print('start position has been set')
        b = time.time()
        with lock:
            payload["switch1"] = True
            payload["switch2"] = False
            payload["direction"] = 1
```

This is the setup part of the simulation function. As can be seen, the function depends on two arguments: lock and payload variables. The meaning of those variables will be explained later. The first line of the code is the assignment of the time of the cycle for simulation. Global variable "tc" holds a value of the time of the cycle in the main part of the code and "t" is the local variable just for the simulation part. Next goes the setup of used GPIO pins. For the mono-stable system simulation, I used one GPIO pin as an input. So in this part I setting all GPIO pins in mode, which is suitable for communication with BCM2835 broadcom chip by command:

```
GPIO.setmode(GPIO.BCM)
```

After that, I need to set up the I01 pin as an input, because of the configuration of JP connectors and with that configuration only I01 and I02 can be used as inputs (more in chapter 4). Due to table 4.2 I01 channel of UniPi is connected to GPIO04 pin of RPi. That‘s why channel 4 is set as input in pulled up mode with the command:

```
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

The next thing is setting up the Relays. Before this setting of the SMBus needs to be done with the command:

```
bus = smbus.SMBus(1)
```

"1" in the arguments of this command means it will setup for newer versions of RPi.

Before explaining next two commands I should explain how runs communication with slave-devices via I2C protocol. I2C is multimaster and multislave device, communication requires several arguments. Typical transaction on writing data with smbus protocol would look like [25]:

```
write_byte_data(addr,cmd,val)
```

Where "addr" is address of the slave device, "0x20" is adress of Relays. A "cmd" is register offset, which means adress of memory zone for writing. Internal memory of the device is divided on several zones, each of them answers for different information. For example register 0x00 ("0" in hex system) is answering for configuration of the Relays [26], and register of the memory 0x09 is answering whether Relays are switched or not. And "val" is value, which will be written in the register. Value is the most important argument in my transaction, because it defines which relays will be switched, when transaction is done. It works that way: there are 8 relays, and range of the value arguments is 2^8. So below is simple table with "val" values for each relay:

-	**Rel 1**	**Rel 2**	**Rel 3**	**Rel 4**	**Rel 5**	**Rel 6**	**Rel 7**	**Rel 8**
Value	128	64	32	16	8	4	2	1

Table 9.1: Values for each relay in UniPi v1.1, [9]

Now returning to the code. I need to specifically configure RELAYS building block (GPA ports) for inputting commands to them by writing them in special register called IODIR [26]. I did it with command:

```
bus.write_byte_data(0x20, 0x00, 0x00)
```

And then I set up starting **initialising** position for simulation. According to 9.2 initializing position is when 1st switcher is on, the 2nd switcher is off, DCV solenoid is on. So simulation on hardware would be: the first relay is switched. I did it by writing "128" as "val" in "0x09" register:

```
bus.write_byte_data(0x20, 0x09, 128)
```

After that, I set up a real-time counter to measure the time of the cycle and delay of the end switchers. Variable "ref" is a reference to the beginning of the cycle and "stime" will be serving as a timer. "b" and "a" variables serving the same purpose but with measuring the delay of the sensors. Delay of end switchers depend on the width of end switchers and velocity of the piston. "a" is a reference variable for the delay, "b" is a real delay timer. So further "while" condition holds the start position. While doing this code writes simulation metadata of starting position in "payload" dictionary variable:

```
payload["switch1"] = True
payload["switch2"] = False
payload["direction"] = 1
```

In this case, the simulator simulates, that the first switcher is pressed, second is off and the piston is directing towards his end position. "1" means "to the end position", "-1" means "to the starting position". On real hardware that means, that 1st relay is on, 2nd relay is off and I01 port has True value.

The next part of the simulation function is the main loop, it does repeat the simulation over and over again and writing metadata about simulated coordinates of the piston for visualization function, so it could draw and visualize it with graph properly.

Description of simulation function with SFC diagram:

Before representing the code of "simulation function", more graphical way of representing this function should be presented. So here on the SFC diagram (fig. 9.1) is shown the sequence of actions, which need to be done to simulate the sequence of actions of the real system on hardware, shown on fig 9.2. In other words, this diagram represents all four states, written in the simulator‘s code in the form of SFC diagram, to show better which conditions lead to which states.

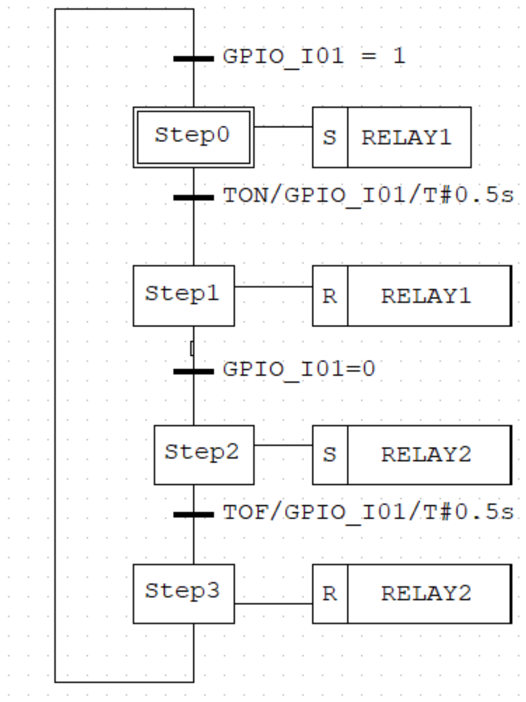


Figure 9.1: Sequence Functional Chart of Actions, Raspberry Pi need to do, to exactly simulate system.

- RElAY1 simulates 1st end switcher (1M1 on figure 2.3)
- RELAY2 simulates 2nd end switcher (1M2 on figure 2.3)
- GPIO I01 simulates 1st solenoid of the bistable DCV (Y1 on figure 2.3)

Condition of transition between 1st state (Step0) and 2nd state (Step1): "TON/GPIO I01/T0.5s" is an expiration of 0.5 seconds from the moment, when GPIO I01 has the logical value True (24 V). Condition of transition between 3rd state (Step2) and (Step3): "TOF/GPIO I01/T0.5s" is also expiration of 0.5 seconds from the moment, when GPIO

I01 has logical value False (0 V). It won‘t always be 0.5 s, because 0.5 s represent the time of reaction of sensors here. So the value can be different. It depends on input parameters (pressure, stroke...).

Description of simulation function with Python code:

Here is the code of it:

```
# main loop
    while alive:
        if not GPIO.input(channel) and (stime - ref < (t-r/2)): # First condition
            bus.write_byte_data(0x20, 0x09, 0)
            stime = time.time()
            with lock:
                payload["switch1"] = False
                payload["switch2"] = False
                payload["direction"] = 1
        elif not GPIO.input(channel) and (stime - ref >= t): # Second condition
            bus.write_byte_data(0x20, 0x09, 64)
            a = time.time()
            b = time.time()
            while (b - a) < r:
                print('end position has been set')
                b = time.time()
                with lock:
                    payload["switch1"] = False
                    payload["switch2"] = True
                    payload["direction"] = -1
            stime = time.time()
        if GPIO.input(channel) and (stime - ref < 2*t-r):  # Third condition
            bus.write_byte_data(0x20, 0x09, 0)
            stime = time.time()
            with lock:
                payload["switch1"] = False
                payload["switch2"] = False
                payload["direction"] = -1
        elif GPIO.input(channel) and (stime - ref >= 2*t):  # Forth condition
            bus.write_byte_data(0x20, 0x09, 128)
            a = time.time()
            b = time.time()
            while (b - a) < r:
                print('start! position has been set')
                b = time.time()
                with lock:
                    payload["switch1"] = True
                    payload["switch2"] = False
                    payload["direction"] = -1
            stime = time.time()
            ref = time.time()
```

The simulated system has 4 states (more in fig 9.2), so the code of the simulator is written to simulate all 4 states. For simulator to transit from one state to another there are conditions for each simulated state:
First condition: According to the figure 9.2 program simulates next state after starting state, which was set before main loop of the function. It simulates transition from start to an end: both switches are off, current going through solenoid of DCV. So hardware simulation of this state is: relays are off, I01 has True value. This state would last time of the cycle. So the condition for this simulation state is:

```
if not GPIO.input(channel) and (stime - ref < t-r/2)
```

"GPIO.input(channel)" command returns True or False value depending on is channel HIGH or LOW. Also, I set this channel in a pulled up note, so the values would be backward. That's checking if the channel is HIGH with "not GPIO.input(channel)" condition. And the second condition says that this state lasts "t - r/2", which means cycle time without end switchers reaction time because they are already included in the loop. Next, the loop turns off relays with the command, which means both switchers are off:

```
bus.write_byte_data(0x20, 0x09, 0)
```

Then renews real-time counter with command

```
stime = time.time()
```

And after that writes current metadata about simulation in the payload variable.

Second condition: Program simulates second state: end position, 2nd end switcher is on, 1st is off, solenoid should be off, this position lasts until the time of the one cycle. Simulation on hardware: 1st relay is off, 2nd relay is on, I01 has False value. So condition is written exactly like that:

```
elif not GPIO.input(channel) and (stime - ref >= t)
```

After this code turn on 2nd relay, which simulates, that second switcher is on:

```
bus.write_byte_data(0x20, 0x09, 64)
```

Setup real delay timer, and wait in that position and simultaneously write metadata in payload variable.

Third condition: Program simulates the third state: a transition state between end position and start position. The difference in condition is that it should last twice as much as the first transition state because real-time cycle counter zeroing out only after it comes back to the first state. The other difference is that there is no current going through the solenoid. Simulation on hardware: relays are off, I01 has False value. So the condition for this state would be:

```
if GPIO.input(channel) and (stime - ref < 2*t-r)
```

All commands after this are the same as in the first condition, except that direction must be "-1" in the metadata variable "payload".

Forth condition: Program simulates starting state, only this time it‘s in the loop. Simulation on hardware: the first relay is on, the second relay is off, I01 has False value. So it requires conditions which look like this:

```
elif GPIO.input(channel) and (stime - ref >= 2*t)
```

DCV solenoid should be off and the time of the cycle should be 2*t. After that code turns on the 1st relay and sets up counter for the delay.

When code checks for all conditions and makes all prescribed actions - it zeroes the real-time counter for the time of the cycle.

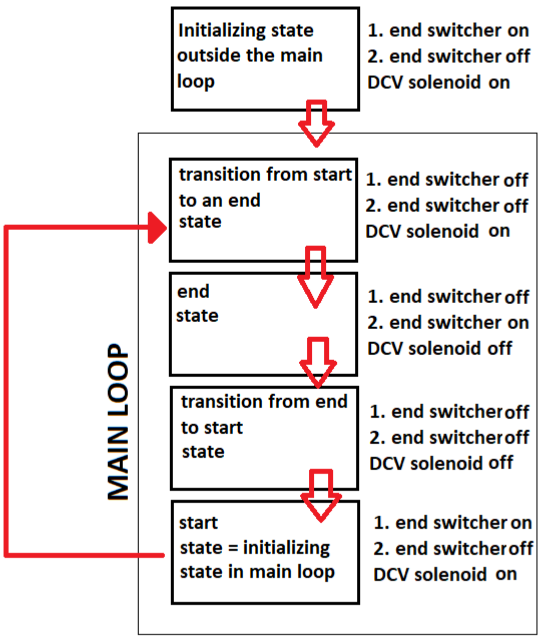


Figure 9.2: Sequence of states of simulated system (monostable valve)

9.1.3 Simulation function for testing on PC

Another function in the script is the "dummy simulation part", which allows testing the simulator on PC. Because of the incompatibility of some libraries with the Windows OS, it’s impossible to run the "simulation part" function and import "SMBus", "RPi.GPIO" libraries on PC with Windows. The main difference with "simulation part" is that it doesn't work with the GPIO ports and Relays, instead, it has declared variables, which simulate the behavior of GPIO ports during simulation:

```
def dummy_simulation_part(lock, payload):
    GPIO = False
```

It has all the same four conditions defined as the "simulation part" function and it writes the same values into the "payload" variable after each condition, but it doesn't depend on GPIO ports presence or values.

Transition "start-end" state:

```
while alive:
        if not GPIO and (stime - ref < (t-r)):
```

End state:

```
elif not GPIO and (stime - ref >= (t-r)):
    .
    .
    GPIO = True
```

At the end of this state "GPIO" variable becomes True, because solenoid should turn off, and originally I have GPIO pins in pulled up state, so the value should be reversed.

Transition "end-start" state:

```
if GPIO and (stime - ref < 2 * (t-r)):
```

Start state:

```
elif GPIO and (stime - ref >= 2 * (t-r)):
```

9.1.4 Visualization function

Third function in the code is Visualization function. It does exactly, what says its name. The core of this function is pygame module. It uses metadata from simulation function to draw objects in the form of a cylinder, piston, and two squares representing end switchers when it's called. It also returns nothing. Now I will go through the code. As it can be seen this function's arguments are also lock and payload:

```
def vizualization_part(lock, payload)
```

The payload is dictionary global variable with metadata. Global means it was defined outside of the function, so it uses this global variable inside the function as an argument. The lock variable will be explained further. After that I define the variable, which contents how much frames per second for this visualization I want and initialize pygame:

```
FPS = 24
pygame.init()
```

After that I set caption of the window and it‘s resolution, which will be containing my visualization:

```
win = pygame.display.set_mode((600, 400))  # define the window with visualisation
pygame.display.set_caption("Visualisation of the pneumatic cylinder")
```

After that I set up method "clock" from pygame to limit FPS to not load RPi too much:

```
clock = pygame.time.Clock()
```

If I won‘t do it, RPi will be trying to draw as much FPS as possible and this is pointless loading for hardware.

The next thing in the code is the definition of coordinates. When the simulator is just started, visualization looks like in figure 9.3.

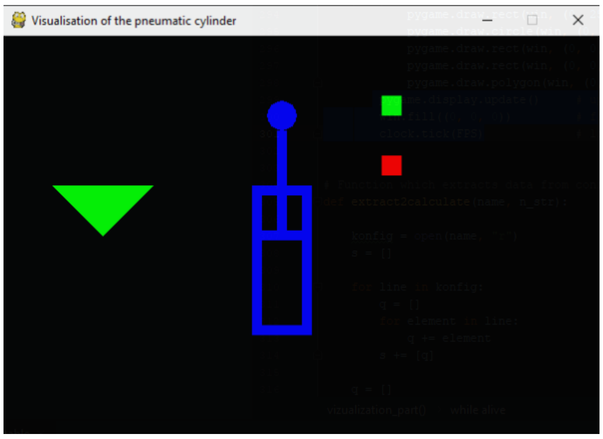


Figure 9.3: Starting position of simulator represented by visualization function

So, because pygame can draw a limited amount of objects: rectangle, square, circle, and line, it must display 10 elements. Those elements are green arrow, left and right walls of the cylinder, floor and ceiling of the cylinder, knob, head and neck of the piston, and two rectangles representing end switchers. Only cylinder and sensors are static in this visualization, which means it doesn‘t move during visualization. For each of the elements, I need to define their disposition on the screen and their sizes. The Center of coordinates of rectangle figures is in their left upper corner. Center of coordinates of circle figures is in their center and for a triangle, it‘s in the center of its bottom. So after defining the FPS and initializing pygame I defined the center of coordinates and sizes of all figures and their relations to each other.

Next goes main loop of drawing. Firstly copy "payload" content with copy function from copy module to "data" variable with this command:

```
while alive:
    with lock:
        data = copy.copy(payload)
```

After this code checks if there is "quit" event in the chain of events of the window with simulation and if so, closes the window:

```
        for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

Next are drawing static objects in "while" loop, what means that those objects are being drawn 24 times per second:

```
# drawing of the static objects:
    pygame.draw.rect(win, (0, 0, 255), (xc1, yc1, wc1, hc1))  # cylinder
    pygame.draw.rect(win, (0, 0, 255), (xc2, yc2, wc2, hc2))  # cylinder
    pygame.draw.rect(win, (0, 0, 255), (xc3, yc3, wc3, hc3))  # cylinder
    pygame.draw.rect(win, (0, 0, 255), (xc4, yc4, wc4, hc4))  # cylinder
    pygame.draw.rect(win, (255, 0, 0), (xs1, ys1, ws1, hs1))  # sensors
    pygame.draw.rect(win, (255, 0, 0), (xs2, ys2, ws2, hs2))  # sensors
```

That leaving code to draw non-static objects. Those are piston, the knob of the piston, and sensors. Each of those objects has two states, but together those non-static objects can be in 6 different states. Transition state with 2 directions: to end position ("1" is written in "payload" variable), to start position ("-1" is written in "payload" variable). Start state, when 1 end switcher is pressed is also with 2 directions and end state when the second switcher is pressed with two different directions. That gives us 6 states together. And those are conditions for them:

- start state with direction "to end position":

```
if data["switch1"] and (data["direction"] == 1):
```

- start state with the direction "to start position":

```
if data["switch1"] and (data["direction"] == -1):
```

- transition state with the direction "to start position":

```
if not data["switch1"] and not data["switch2"]:
    if data["direction"] == 1:
```

- transition state with the direction "to end position":

```
if not data["switch1"] and not data["switch2"]:
    else:
```

- end state with direction "to end position":

```
if data["switch2"] and (data["direction"] == 1):
```

- end state with direction "to start position":

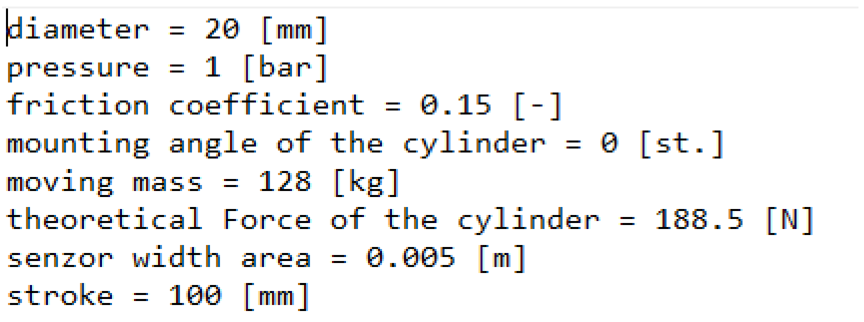
```
if data["switch2"] and (data["direction"] == -1):
```

Depending on the condition pygame draws different positions for those objects. Green arrow changes its direction depending on the position: down - for direction "to end position", up - for direction "to start position" (more at figure 9.2). The lower red rectangle represents the first end switcher and the upper represents second end switcher. Then they are "pressed" in the simulation, rectangles change their colors to green. And At the end of the visualization part, there are methods from the pygame module. That update and fill the window with black, because every time pygame changes the picture and that is how animation is made, it fills new pixels with different colors according to the program, but it doesn't erase previous pixels. That's why the window updated and then filled with black color. Also in the end there is a method, which is limiting the FPS for visualization. Code looks like this

```
pygame.display.update()
win.fill((0, 0, 0))
clock.tick(FPS)
```

9.1.5 Collecting data

There is two more written function in code, one of them is "extract2calculate" function. It helps to extract parameters of the cycle from the configuration .txt file. Simulator extracts data about simulation, which includes working pressure of cylinder, a stroke of the cylinder, diameter of the cylinder, a mass of the piston, and width of the sensors. Then code calculates the time of the cycle from that data and creates the simulation of the cycle with this exact length, calculation of the time happens in the main code. Configuration file looks like this 9.4.



```
diameter = 20 [mm]
pressure = 1 [bar]
friction coefficient = 0.15 [-]
mounting angle of the cylinder = 0 [st.]
moving mass = 128 [kg]
theoretical Force of the cylinder = 188.5 [N]
senzor width area = 0.005 [m]
stroke = 100 [mm]
```

Figure 9.4: Example of configuration file

The function is called "extract2calculate" and has two arguments, name of the configuration file and, number of the line, from which it will extract the number. Firstly code opens configuration file in reading mode, that‘s why it has "r" as an argument and creates empty list for extracted data:

```
def extract2calculate(name, n_str):
    konfig = open(name, "r")
    s = []
```

After this I extract every symbol from configuration file in two-dimensional list variable "s":

```
    for line in konfig:
        q = []
        for element in line:
            q += element
        s += [q]
```

Then I nulify "q" for reusing. And checking every symbol in "s" if it‘s number or dot. Then I put numbers and dots in "q" two-dimensional list:

```
    for i in range(len(s)):
        for j in range(len(s[i])):
            q += ['']
            if s[i][j].isdigit() or (s[i][j] == "."):
                q[i] += s[i][j]
```

And in the end function returns number from certain line in "float" format:

```
    return float(q[n_str])
```

9.1.6 Plot function

This function is visualising coordinate of the piston and logical value on both switches. The core of the function is submodule "pyplot" from "matplotlib" module. The graph is synchronised with visualisation function through "payload" variable. Function returns nothing, but drawing the graph.

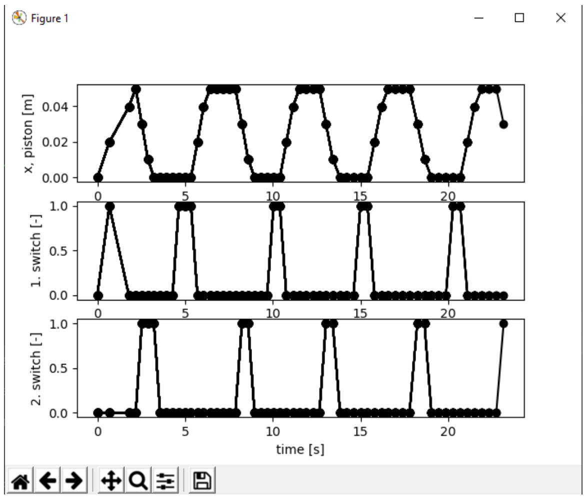


Figure 9.5: Visualisation of the simulation through the plot

So, at the beginning of the function there is declaring of the function itself and defining lists for writing the values for the graph in it.

```
def plot_viz(lock, payload, vel, reference):
    y = [0]
    x = [0]
    ys1 = [0]
    ys2 = [0]
```

Function is depending on "lock" and "payload" variables as "simulation" and "visualisation" functions. After this goes the main loop of the function.

```
    while alive:
        a = time.time()
        with lock:
            if payload['direction'] == 1:
                if y[-1] >= stroke - vel:
                    y += [stroke]
                else:
                    y += [y[-1] + vel]
            elif payload['direction'] == -1:
                if y[-1] > 0 + vel:
                    y += [y[-1] - vel]
                else:
                    y += [0]

            if payload['switch1']:
                ys1 += [1]
            else:
                ys1 += [0]
            if payload['switch2']:
                ys2 += [1]
            else:
                ys2 += [0]
```

At the beginning of the main loop I define the time of the beginning of every cycle in variable "a". After that I write the coordinates of the piston by adding or subtracting the velocity from previous coordinate. If direction is "1", then I add, because direction is positive and otherwise I subtract. Also depending on the state of the switches I write "0" or "0" in the list of their logical values. Condition are working with data from "payload" variable, which is changing in "simulation" function. Now continuing with the main loop of the function:

```
    x += [a - reference]
    plt.subplot(311)
    plt.plot(x, y, '-ok')
    plt.ylabel('x, piston [m]')
    plt.subplot(312)
    plt.plot(x, ys1, '-ok')
    plt.ylabel('1. switch [-]')
    plt.subplot(313)
    plt.plot(x, ys2, '-ok')
    plt.xlabel('time [s]')
    plt.ylabel('2. switch [-]')
    if (a-reference) > 20:
        del y[0]
        del x[0]
        del ys1[0]
        del ys2[0]
    plt.plot()
    plt.pause(0.25)
    plt.clf()
```

After all the conditions I write the time in variable "x" by subtracting the "reference", which was made in the beginning of the main code, from "a" , which was made on the beginning of the loop. Then script draws plot with three subplots: first - coordinate of piston from time, second - logical value of first switch from time, third - logical value of second switch from time. Next goes if condition: if 20 seconds have past from the beginning of the loop, then script deletes old values from "y, x, ys1, ys2" lists. Command "plt.plot()" shows the graph and command "plt.pause(0.25)" renews it every 0.25 seconds. And "plt.clf()" redraws the figure, so the graph shows only last 20 seconds of the simulation.

Important remark: While I was testing the simulator with visualization function and plot function included on PC, simulation went just fine. But at testing on RPi plot function started working improperly. Because of the limitation of the processor of RPi and because of function is getting metadata from the same variable "paload" as "visualization part" function renewed plot every 50-60 seconds and graf didn't make sense. "visualization part" function worked great, just as it would work without "plot viz" function included. Because of "payload" variable is being opened by those function with lock, which means, that while one function using variable, other can't use it, "plot viz" function is getting access to the "payload" relatively rarely, that may be the cause of improper work. The solution here is more powerful hardware (later versions of RPi) or optimalization of the function on level, which I'm right now incapable of.

9.1.7 Main code

This is the part of the code, where all the above-mentioned functions are run and all global variables, which are used in those functions, are being declared. Main code starts with declaring the variable "alive", that serves as main switcher of the whole simulator and also declaring the beginning of the cycle in the variable "reference" for "plot viz" function:

```
alive = True
reference = time.time()
```

Then I declare the "payload" variable, which is serving as metadata, written in simulation function and used in visualization function:

```
payload = {
    "switch1": False,
    "switch2": False,
    "direction": 0
}
```

After that goes assignment of the name of the configuration file to the variable "config file" and then extraction of the parameters of the cycle with help of "extract2calculate" function and assignment of its values to the corresponding variables:

```
config_file = "konfiguration.txt"
d = extract2calculate(config_file, 0)/1000
p = extract2calculate(config_file, 1)*10**5
mu = extract2calculate(config_file, 2)
alpha = extract2calculate(config_file, 3)
m_m = extract2calculate(config_file, 4)
```

```
F_th = extract2calculate(config_file, 5)
s_s = extract2calculate(config_file, 6)
stroke = extract2calculate(config_file, 7)/1000
```

"d" is the diameter of the cylinder translated from mm to meters, "p" is working pressure translated from bars to Pascals. "mu" is friction coefficient for steel-steel. "alpha" is the mounting angle of the cylinder. "m m" is mass, that cylinder moves. "F th" is a theoretical force, that cylinder can apply. "s s" is the width of the sensors. "stroke" is a stroke of the cylinder translated from mm to meters. All those data can be found in the standard specification of the pneumatic cylinder. In my case, I used the "DSNU-20-100-PPV-A" cylinder as an object for simulation.

Next goes the calculations of the time of the cycle from the parameters of the cycle. The calculation is trivial. The scheme is displayed on figure 9.6.

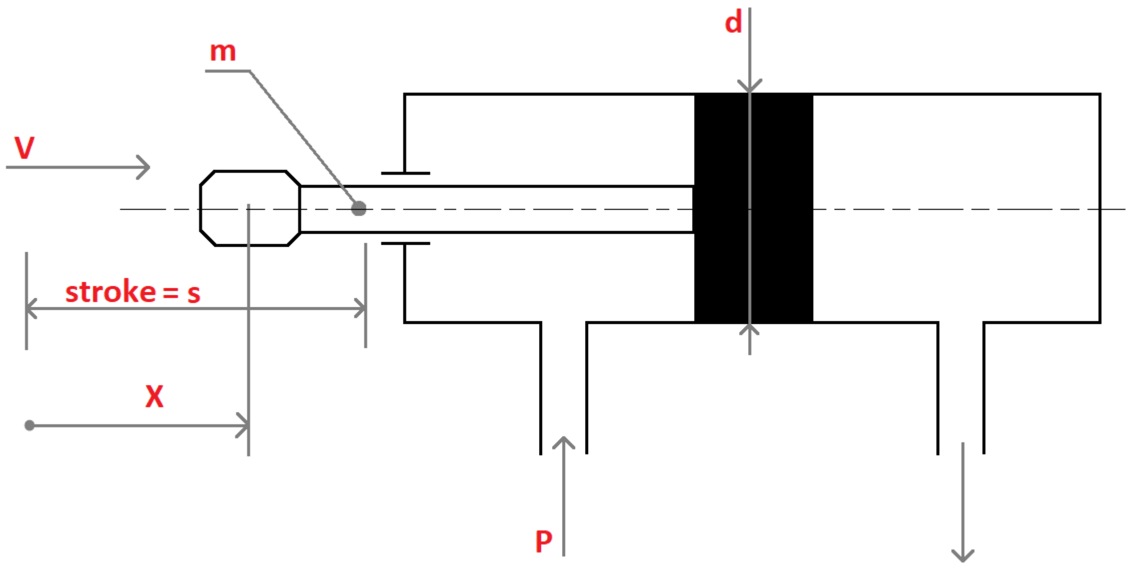


Figure 9.6: Scheme for calculation

I took principals of calculations from Festo Fluidsim 5 software official tutorials [27] (calculations are for cylinder working with specific load) and wrote calculation based on it.:

$$a = \frac{p \cdot A}{m_m} \tag{9.1}$$

$$A = \frac{\pi \cdot d^2}{4} \tag{9.2}$$

, where "A" - area of the head of the piston, "p" - working pressure, "m m" is mass, that piston moves "moving mass" or a load of cylinder, "d" is the diameter of the cylinder.

Calculation of the velocity of the piston and time of actuation:

$$stroke = \frac{1}{2} \cdot t^2 \cdot a \tag{9.3}$$

$$t = \sqrt{\frac{2 \cdot stroke}{a}} \tag{9.4}$$

$$v = t \cdot a \tag{9.5}$$

Condition for the piston to start moving:

To start moving, the piston should overcome the static force of friction. Basically theoretical force of piston "F theor" should be greater than "F b". "F theor" can be found in the documentation of a certain cylinder.

$$F_B = \mu \cdot m_m \cdot g \cdot Cos(\alpha) \tag{9.6}$$

$$F_{theor} > F_B \tag{9.7}$$

Where "mu" is friction coefficient, "m m" is moving mass (load) and "alpha" is the mounting angle of the cylinder.

And from that it‘s possible to calculate **the time of reaction of the sensors:**

$$t_{reaction} = \frac{2 \cdot s_s}{v} \tag{9.8}$$

,where s is the width of the sensors.

So next lines are calculating the acceleration, time of the cycle, velocity of the piston, static friction force, and time of reactions according to 9.4, 9.5, 9.6, and 9.8 expressions.

```
A = (math.pi*d**2)/4
acc = (p*A)/m_m
tc = math.sqrt((2*stroke)/acc)
vel = tc * acc
r = (2*s_s)/vel
F_b = mu*m_m*9.81*math.cos(alpha)
```

After that comes implementation of condition of overcoming the static friction force and printing the results of calculation:

```
if F_th < F_b:                           # condition: if moving mass is too big
    vel = 0
    tc = 100000000
    r = 10000000
```

```
print(f'Calculated time of the cycle: {tc} [s]')
print(f'Calculated velocity of the piston: {vel} [m/s]')
print(f'Calculated reaction time of the sensors: {r} [s]')
```

The next thing is the condition: when code has errors, it most likely is because it‘s run on windows and code couldn’t import "SMBus" and "GPIO" library. In that case, code needs to be at least tested on windows, that‘s why code replace "simulation part" function with "dummy simulation part" function, which is made especially for testing on windows:

```
if DEBUG:
    simulation_part = dummy_simulation_part
```

After this code defines an object "lock", which is useful when using threading parallelism. It‘s locking global variables, so two threads couldn’t access the same variable at the same time, that would cause an error:

```
lock = threading.Lock()
```

So that‘s why this object "lock" is in arguments of simulation, visualization and "plot viz" functions. Next is defining the threads and starting them, from now on they will be running in parallel:

```
thread_simulation = threading.Thread(target=simulation_part, args=[lock, payload])
thread_vizualization = threading.Thread(target=vizualization_part, args=[lock, payload])
thread_plot = threading.Thread(target=plot_viz, args=[lock, payload, vel, reference])

thread_simulation.start()
thread_vizualization.start()
```

And in the end is "main switcher" loop. The simulator will be running, except "Ctrl+C" combination is pressed on the keyboard:

```
print('starting the process')
while True:
    try:
        time.sleep(0.01)
    except KeyboardInterrupt:
        print('Total Stop')
        alive = False
        sys.exit()
```

9.1.8 Difference between simulating the system with monostable valve and bistable valve

First difference is that in "simulation part" function I need to define two channels for two inputs, because bi stable system have two solenoids, which code needs to simulate:

```
print('starting the process')
GPIO.setup(channel_1, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(channel_2, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Description of simulation function with SFC diagram:

The main difference between simulator for the system with bistable and monostable valve is the main loop in the "simulation part" function, so before source code will be presented, more graphic representation should be shown. That would be SFC diagram from the side of Raspberry Pi.

Here on the SFC diagram (fig. 9.7) is shown sequence of actions, which are simulating the sequence of actions of the real system, shown on fig 9.8. In other words, this diagram represents all four states, written in the simulator‘s code in the form of SFC diagram, to show better which conditions lead to which states.

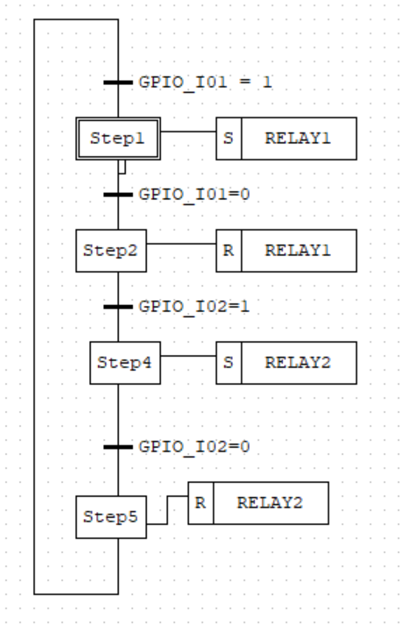


Figure 9.7: Sequence Functional Chart of Actions, Raspberry Pi need to, do to exactly simulate system.

- RElAY1 simulates 1st end switcher (1M1 on figure 2.6)
- RELAY2 simulates 2nd end switcher (1M2 on figure 2.6)
- GPIO I01 simulates 1st solenoid of the bistable DCV (Y1 on figure 2.6)

- GPIO I02 simulates 2nd solenoid of the bistable DCV (Y2 on figure 2.6)

Description of simulation function for the bistable system with code in Python:

Respectively in the "simulation part" function, there will be other conditions for changing the metadata in "payload" for the visualization part. I added the extra condition to separate the cycle on the "start-to-end" part and "end-to-start" part by adding the "add cond" variable and making it "True" at the beginning of the function and changing it through the main loop of the function. It was done because both transition "end-start" and "start-end" states have the same combination of inputs (more on fig. 9.8) and output should be different. In other words: it's a sequential system and it requires additional input. The first "monostable" system simulator is separated the same way by the logical value on the DCV solenoid, but in "bistable" system simulator there isn't such value because of the solenoids are being turned on the small amount of time, therefore I used "add cond" variable:

```python
add_cond = True
```

The sequence of conditions execution inside the main loop of simulation function for a system with a bistable valve is presented below for better understanding in figure 9.8.

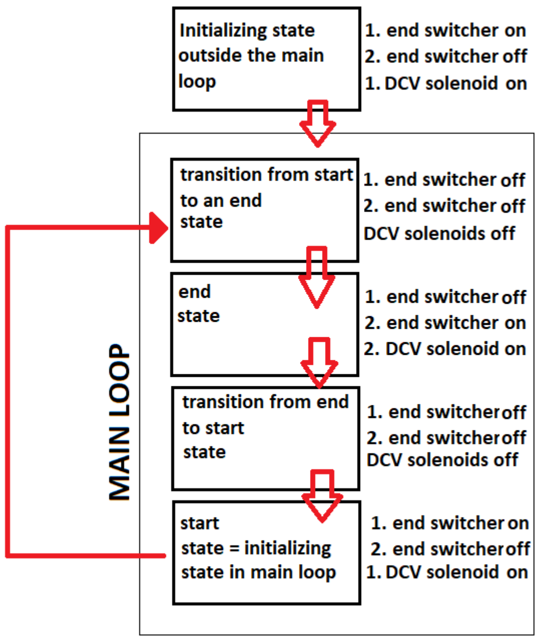


Figure 9.8: Sequence of states of simulated system (bistable valve)

First condition: Program simulates transition state from start to an end, both switchers are not pressed during the state, both solenoids are off. So simulation on hardware is: both relays are off, I01 and I02 have False value, but at the beginning, I01 is on. There is a current going through the first solenoid out of two (Y1 on figure 2.6), but only at the beginning of the cycle, when the first switcher is pressed. After this everything is off:

```
        if stime - ref < (t-r/2):
            if not GPIO.input(channel_1):
                bus.write_byte_data(0x20, 0x09, 0)              # turn off relays
```

So there is the condition for the full transition state and sub condition for the beginning.

Results of this condition would be the same as in mono stable simulator:

```
with lock:                                    # metadata about the sim for vis
    payload["switch1"] = False
    payload["switch2"] = False
    payload["direction"] = 1
```

Second condition: Program simulates end state, 2nd solenoid is on, second switcher should be turned on. Simulation on hardware: 2nd relay is on, I02 has logical True. So condition is:

```
        elif add_cond and (stime - ref >= t):
            bus.write_byte_data(0x20, 0x09, 64)
            add_cond = False
```

Here ends the "start-to-end" part of the simulation and the "end-to-start" part begins. So "add cond" becomes false.

And results:

```
with lock:              # metadata about the simulation for visualization
    payload["switch1"] = False
    payload["switch2"] = True
    payload["direction"] = -1
```

Third condition: Program simulates transition state from end to start, both switchers are not pressed during the state, but the second switcher is pressed at the beginning. There is a current going through the second solenoid out of two (Y2 on figure 2.6), but only at the beginning of the cycle, when the second switcher is pressed. After this everything is off. So simulation on hardware is: both relays are off, I01 and I02 have False value, but at the beginning, I02 is on:

```
        if (stime - ref < 2 * t-r) and not add_cond:
            if not GPIO.input(channel_2):
                bus.write_byte_data(0x20, 0x09, 0)                      # turn off relays
```

That's why I needed "add cond" variable because "stime-ref" at the beginning of the cycle falls under two conditions simultaneously, it's less than (t-r) and less than 2*(t-r), to separate it I added "add cond"

Results:

```
with lock:                # metadata about the sim for viz
    payload["switch1"] = False
    payload["switch2"] = False
    payload["direction"] = -1
```

Forth condition: Program simulates starting condition. Second solenoid is off, First switcher should become on, second is off. Real time counter nullifies as the result. Simulation on hardware: 1st relay is on, I01 has logical True. Condition expires after time of the cycle is over:

```
        elif stime - ref >= 2 * t:
            bus.write_byte_data(0x20, 0x09, 128)
```

Results:

```
with lock:  # writing the metadata about the simulation for visualisation
                payload["switch1"] = True
                payload["switch2"] = False
                payload["direction"] = -1

ref = time.time() #nullify the counter
```

Important remark: The code of simulator for the system with a bistable valve, that wasn't mentioned in this subchapter remains the same as in simulator for the system with monostable valve.

9.2 Developing in NodeRed

Node-RED [21] provides a web browser-based flow editor, which can be used to create JavaScript functions. Elements of applications can be saved or shared for re-use. The runtime is built on Node.js. The flows created in Node-RED are stored using JSON. So, in my case node-red is running on RPi, and I creating flows via an internet browser on an external computer. The computer communicates with RPi via TLS protocol. RPi creates a localhost server. Basically, to use node-red on RPi + UniPi – it's necessary to run node-red on RPi, which is connected to the internet and go access RPi via browser in the local network.

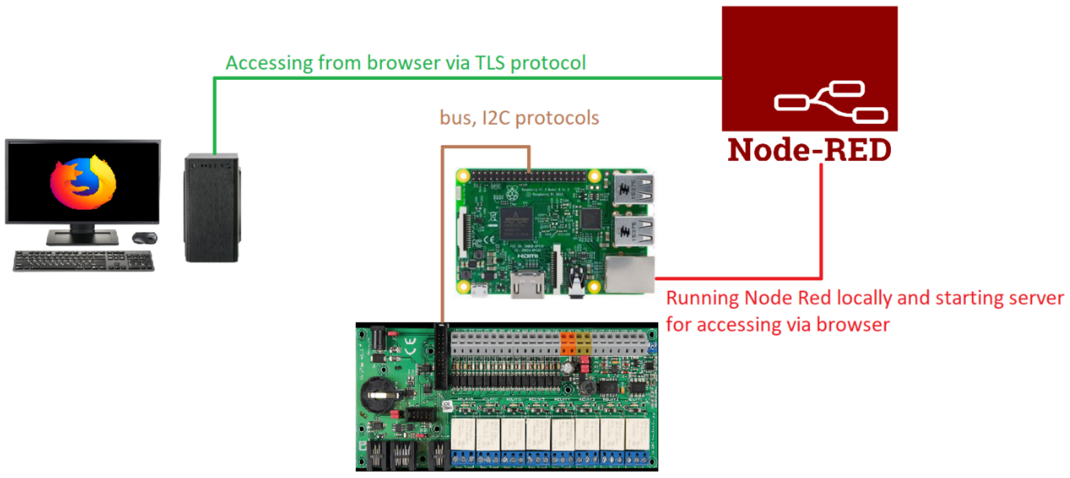

Accessing from browser via TLS protocol
bus, I2C protocols
Node-RED
Running Node Red locally and starting server for accessing via browser

Figure 9.9: Scheme of connection between node red and RPi

Node-RED is a flow-based development tool for visual programming. Nodes could be understood like specific purpose functions. The simplest flow it is possible to create with nodes would look like this:

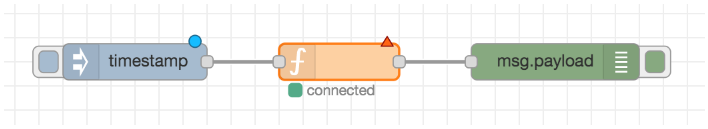


Figure 9.10: Simplest flow in Node Red

The first blue node on the left is inject node, which by default deploys timestamp to the msg object with payload property when the button on the left of it is pushed. However, it is possible to change the information, that the inject node deploys to msg.payload. It could be a string like “Hello world!”, number, or a Boolean. The next orange node is function node, default function just returns msg.payload without any changes, but you can change it by writing JavaScript function inside of this node. The important thing is that you can make changes to msg.payload, but it should always return msg as a whole object or with some property at the end of the function. And the last one is the green debug node. By default it prints msg.payload on the screen, you can make it print the whole msg objects or some other properties of msg.

9.2.1 Created simulator in Node-Red.

Flow in simulator program is displayed on figure 9.11. It works that way: There are two branches in this flow. The lower one functions as a switch of the main upper branch. The “set gate” yellow node is playing the role of a global object. Violet "Enable" node is writing the True to property msg.payload. Violet "Disable" node writes false to msg.payload. The yellow changing node "set gate" is writing a value from msg.payload object to flow.gate object. And if "True" value is written to "set gate" node - yellow node "gate" in upper "main" branch works as bypass, if "False" value is written to "set gate" node - yellow node "gate" in the upper main branch doesn't work as bypass, so upper main branch doesn't work.

And the upper one is the **"Main" branch**. Now you can push the "start" violet node in the upper main branch, it will inject "start" into function node. The algorithm of the yellow function node "logical controller" is:

- if you inject "1", it will inject "1" in the first (upper) output
- if you inject "0", it will inject "1" in the second (middle) output
- if you inject "start", it will inject "1" in the third (lower) output

That's because there are written Java Script function in it:

```
if (msg.payload == 1){
    msg.payload = 1;
    return[msg, null, null]
}
if (msg.payload == 0){
    msg.payload = 1
    return[null, msg, null]
}
if (msg.payload = "start"){
    msg.payload = "start"
    return[null, null, msg]
}
```

When "start button" is pressed and "start" is bypassed by "gate" node and injected into "logical controller" function, yellow function node will inject "start" in third output, it will pass "1" to red "i2c" node and green debug node, which will indicate, that cycle is started, by printing "start" in debug window. Red "i2c" node will switch the first relay. When the first relay is switched, PLC program will make input (PIN 11) True and from now on node "PIN 11" is input to the logical controller. So, it will pass "0" to the yellow function node (because it is in "pull-down" mode). The function node will pass "1" to the second output and after a delay time, the second relay will be switched. After that PLC program will make input False, so the blue "rpi gpio in" node will pass "1" (because it is in "pull-down" mode) to the yellow function node. The function node will pass "1" to the first output and after delay time the first relay will be switched.

There are two "i2c" nodes and delay nodes on two first outputs from the "logical controller" node. It‘s because I want the relay to switch on after a certain time (time of the cycle) and switch off after a certain time (reaction time of sensors). That‘s why I made it that way. When function node passes “1” to the first or second output, it passes it to the “time of cycle” branch and on the “reaction time of sensors” branch, so actually “reaction time of sensors” delay = “time of cycle” delay + “actual reaction time, that I want in this simulation”. So, both delays start at the same time, only the “reaction time of sensors” delay is longer because we want our relay to switch off after reaction time.

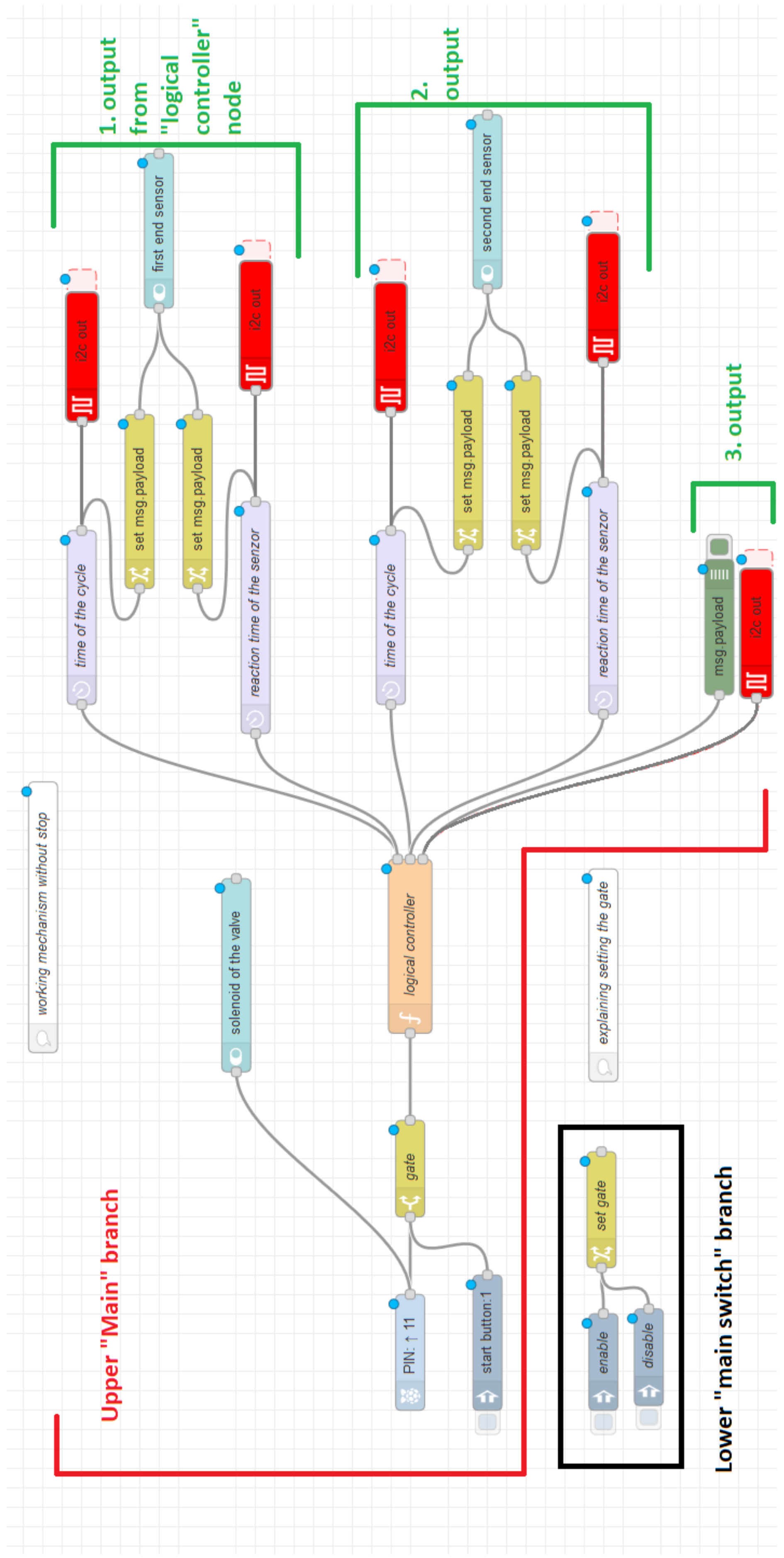


Figure 9.11: Simulator in Node Red

Visualization of this simulator is implemented way simpler than in the Python-based simulator. With the help of a special module called "Node-Red-Dashboard" - it's possible to create a visualization of data from sensors on the webpage, which is run locally on RPi. There weren't suitable instruments for visualizing states of the simulated solenoid and two simulated end switchers, like a bulb or something like this. But there was a possibility of making the switch with inverted function i.e. switcher is displayed switched, when for example current is going through the simulated solenoid, or any of simulated end switchers is switched.

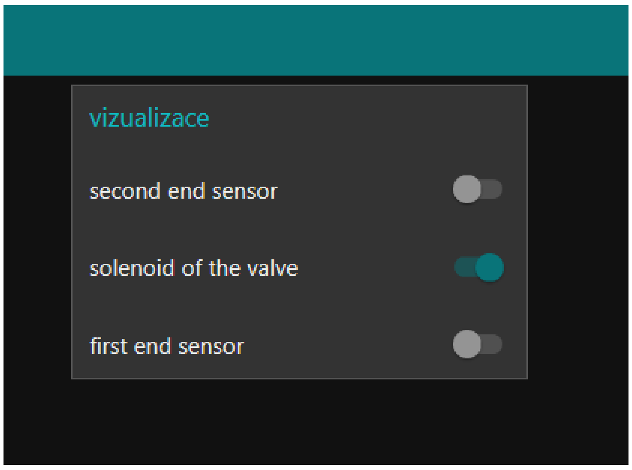


Figure 9.12: Visualization in Node Red

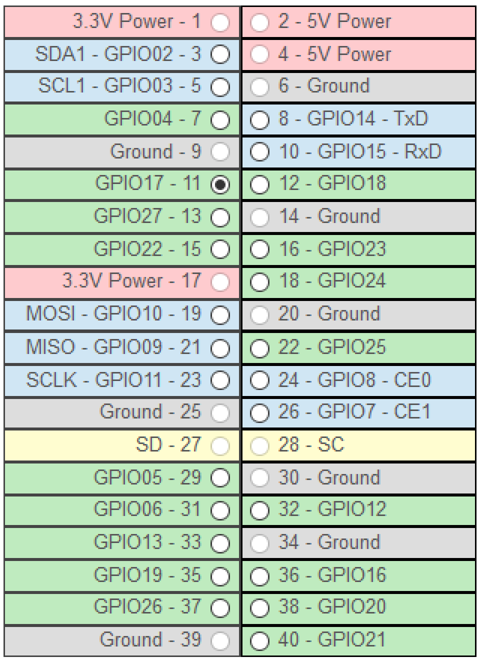
The example of the visualization can be seen in figure 9.12. In the state shown on the figure 9.12 - current is going through the simulated solenoid and both end switchers are off. So this is the transition state of going from the start position to the end position.

The switches are switching because of light blue nodes, called "switch" are connected to input "PIN: 11" and to outputs from the "logical controller" node.

9.2.2 Used Nodes and Modules

RPi gpio in node: This is additional node from "node-red-node-pi-gpio" module. It's necessary to download this module before using it. This node checks the input on RPi and depending on "pull-up" or "pull-down" mode deploys 1 - "pull-up", 0 - "pull-down", when input is HIGH and backward when input is LOW. In my case node checks on pin №11, which is GPIO 17 on RPi according to the table 9.2. And GPIO 17 is I02 on UniPi v1.1 according to table 4.2. I chose I02 port instead of I01 port, because of bugs with Node-Red on RPi v1.2.

Figure 9.13: rpi-gpio in node



3.3V Power - 1 ○	○ 2 - 5V Power
SDA1 - GPIO02 - 3 ○	○ 4 - 5V Power
SCL1 - GPIO03 - 5 ○	○ 6 - Ground
GPIO04 - 7 ○	○ 8 - GPIO14 - TxD
Ground - 9 ○	○ 10 - GPIO15 - RxD
GPIO17 - 11 ●	○ 12 - GPIO18
GPIO27 - 13 ○	○ 14 - Ground
GPIO22 - 15 ○	○ 16 - GPIO23
3.3V Power - 17 ○	○ 18 - GPIO24
MOSI - GPIO10 - 19 ○	○ 20 - Ground
MISO - GPIO09 - 21 ○	○ 22 - GPIO25
SCLK - GPIO11 - 23 ○	○ 24 - GPIO8 - CE0
Ground - 25 ○	○ 26 - GPIO7 - CE1
SD - 27 ○	○ 28 - SC
GPIO05 - 29 ○	○ 30 - Ground
GPIO06 - 31 ○	○ 32 - GPIO12
GPIO13 - 33 ○	○ 34 - Ground
GPIO19 - 35 ○	○ 36 - GPIO16
GPIO26 - 37 ○	○ 38 - GPIO20
Ground - 39 ○	○ 40 - GPIO21

Table 9.2: Node GPIO module pin configuration

Switch node: allows messages to be routed to different branches of flow by evaluating a set of rules against each message [28]. Used in flow as a bypass from input "PIN 11" node to function node.

Figure 9.14: Switch node

Change node: Set, change, delete or move properties of msg, flow such as msg.payload or flow.gate. I used it for moving or “redirecting” True or False value from “enable” or “disable” buttons from lower brunch to switch node “gate” from upper main brunch. I wrote it in msg.payload and then moved it from msg.payload to flow.gate

Figure 9.15: Change node

Function node: I've described this node in upper example with the simplest flow

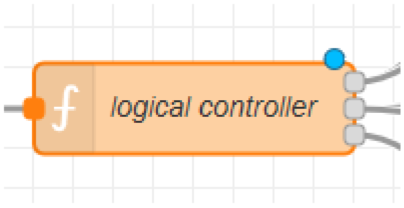


Figure 9.16: Function node

Delay node: This node delays the passing of the message. You can change the time of the delay by changing the argument of the node.

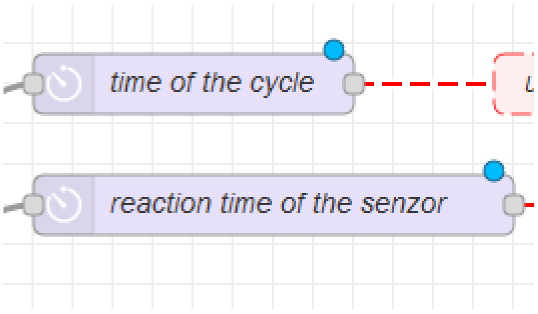


Figure 9.17: Delay node

I2C node: This is additional node from "node-red-contrib-i2c" module. It's necessary to download this module before using it. This node allows to control building blocks connected with I2C protocol on RPi + UniPi. In our case, I used it to control relays. Node has 3 arguments: slave device address: 0x20 for the relays, sending bytes: 1 for relay to switch, and 0 to do nothing. And payload: -128 for the first relay to switch, 64 for the second, 32 – 3rd, 16 – 4th, And so on till 1 is for the 8th relay.

Figure 9.18: i2c node

Switch node: can set the icon of the switch in the dashboard layout, depending on state [29].

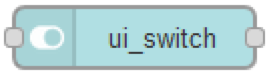


Figure 9.19: UI switch node

9.3 Developing in Mervis

Mervis [23] is a software platform for control of UniPi programmable controllers. Mervis is a set of tools and applications for advanced regulation and automation. You can write program in FBD or ST methods. I used Function Blok Diagram (FBD). To use Mervis with UniPi v1.1, it's firstly necessary to burn the image of Mervis‘s version of Raspbian (Mervis OS) on the SD card and install Mervis IDE on your computer.

Mervis OS is a specially configured version of Raspbian without a graphical interface. Controlling the system is possible through the command line, but the most convenient way to do this is through Mervis IDE.

9.3.1 Preparing Mervis

To make Mervis work it‘s necessary to connect UniPi and computer with Mervis IDE to the same local network.

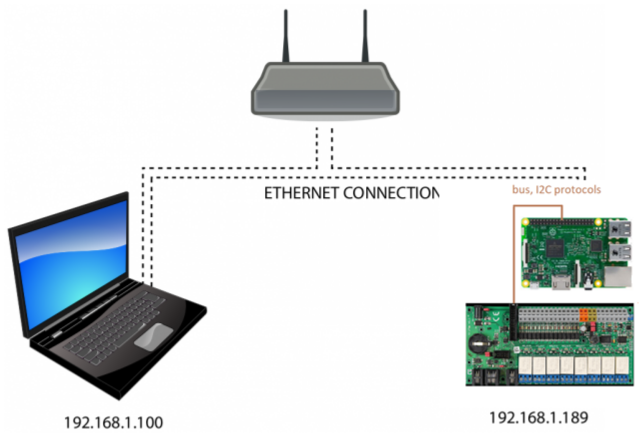


Figure 9.20: Connection of Mervis and RPi+UniPi

After It ‘s connected to the same local network, it‘s required to attach Mervis IDE to the controller. The next step is configuration inputs into outputs. To connect to the UniPi, it's needed to add Channel. The Channel is a communication point with hardware that happens in a defined protocol. Next thing - to change is the protocol to Modbus. The Channel is configured, now the next step is to add the device. Device should be UniPi v1.1 [30]. After all channels and devices are configured, developing of the program can be started.

9.3.2 Created simulator in Mervis IDE

After the device is configured, a list of all inputs and outputs will appear in Mevis IDE. Those inputs and outputs occur as variables, which contain the current state of selected output or input. But you can add to the list your variables, which will represent different things. For example, my program in Mervis IDE contains the time of the cycle as a variable.

Now the following list is the **list of variables and functions, used in program:** List of variables:

- UniPi1-AO-DI-CNT DI01 – current value of Digital Input 1, Boolean
- time ton – delaying time for TON function, Real
- time tof – delaying time for TOF function, Real
- UniPi1 Relays Relay1 – the value of Relay 1, Boolean, 0 – Relay is in starting position, 1 – end position
- UniPi1 Relays Relay2 – the value of Relay 2, Boolean, 0 – Relay is in starting position, 1 – end position

List of functions:

- NOT – The function negates the input
- TON – The delaying timer of the rising edge of the input signal
- TOF – The delaying timer of the falling edge of the input signal

Variables are represented as cells in grey columns with the name of the selected variable on the left and on the right of the figure 9.21. Variables on the left are input in the program, the output from the program writes in the column with variables on the right. On the left, in both branches, there is DI1 GPIO port as input. As output, there are RELAY 1 in the upper branch and RELAY 2 in the lower branch.

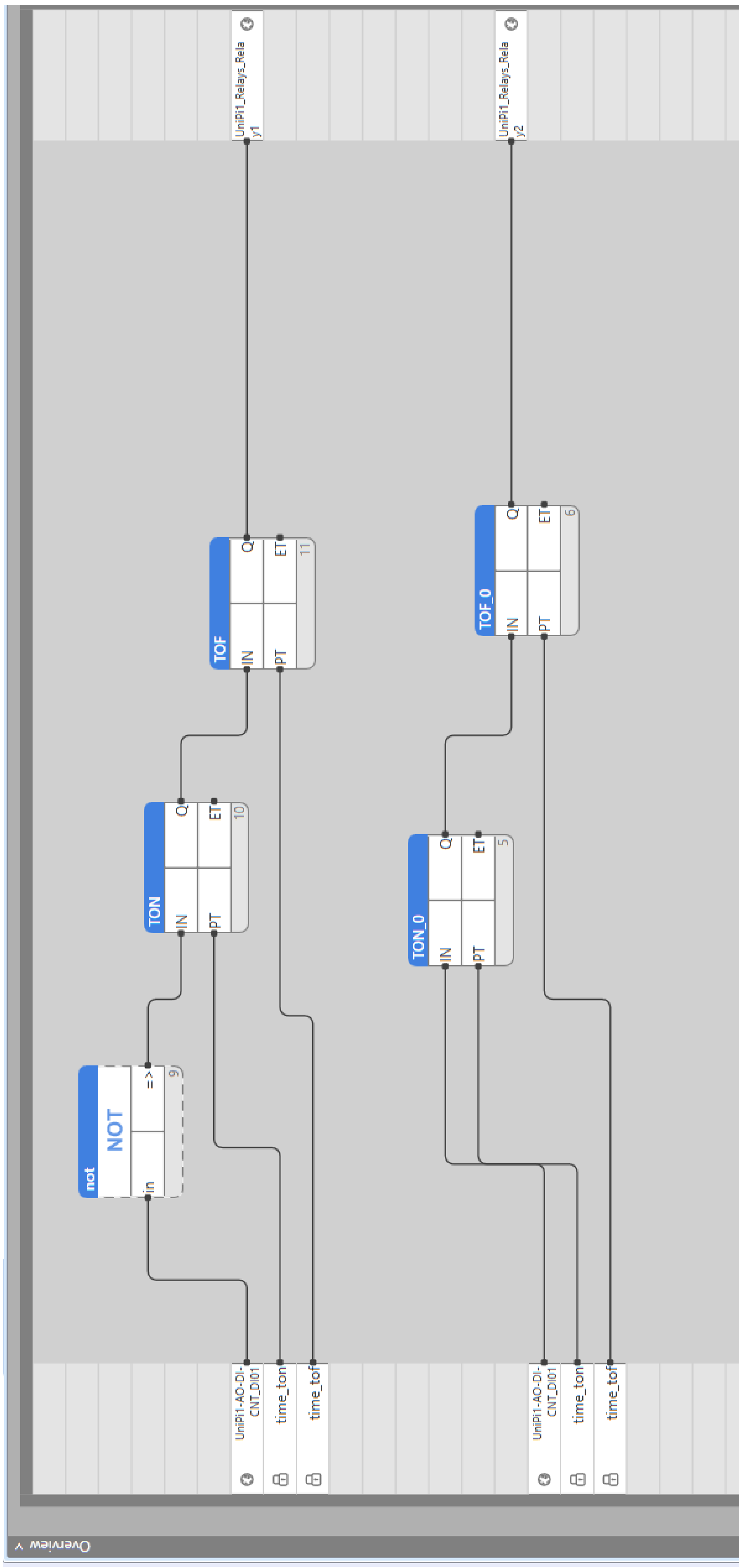


Figure 9.21: Simulator in Mervis IDE

Now principle of the simulator. When Digital Input 1 gets HIGH, logical “1” passes from "UniPi1-AO-DI-CNT DI01" through both branches (upper and lower), passes through a delay of the rising edge and through a delay of the falling edge. Only in the upper branch, it passes also through NOT function, which negates the value. So, at the end logical “1” is on the "UniPi1 Relays Relay2" variable. And logical “0” is on the "UniPi1 Relays Relay1" variable. While Digital Input is LOW – values of Relay 1 and Relay 2 would be another way around. I had used two functions of delays TON and TOF to get delay for “turning on” and “turning off” the relays. Because each one of them delaying only edges of the signal and not the whole width of the signal. That‘s why variables "time ton" and "time tof" should be equal, to delay the whole signal on the same amount of time. Also the value of "time ton" and "time tof" should include not only the time of the cycle but also the reaction time of the sensor. So if you want to make a simulation with the time of the cycle = 5 [s] and with the reaction time of the sensor = 0.5 [s], the value of "time ton" and "time tof" should be 5.5 [s].

Mervis IDE allows it’s users to make HMI with special HMI editor. This editor lools like the layout of elements, which can be interactive like switchers or can display information. These elements allow you to visualize the status of your system and control all of its functions. For this purpose, the editor features a large collection of switches, indicators, text fields, time schedulers, and other elements [24]. I used this HMI editor to make visualization of the process of the simulation.

The visualization of the simulator in Mervis IDE very much looks like visualization in Node-Red (More on figure 9.22). It has three indicators with the title. Each represents one of the switchers or solenoid of the valve. Each shows if it’s True or False.

OFF
First end switcher
OFF
Second end switcher
OFF
Solenoid of the valve

Figure 9.22: Visualization of simulator in Mervis IDE

Chapter 10

Conclusion

The task of this bachelor thesis was:

- To get acquainted with RPi programming
- Connect UniPi I/O module to RPi
- Design and test the way of simulation of selected ones technological elements using several platforms (Python, Mervis ...)
- Test the functionality of the simulator by connecting it to the control PLC

So in the theoretical part of this bachelor thesis, I described how I had become acquainted with hardware and methods of communication between different pieces of the hardware, which is necessary for becoming acquainted with the programming of the RPi. In the practical part, I've described the development of the simulator based on three different platforms on RPi+UniPi hardware. Therefore, while I was developing those simulators, I got acquainted with the programming of RPi with Python, which is the main programming language for it, with Node-Red visualization software and with Mervis IDE.

For testing of the developed simulator, I've connected UniPi with RPi and described in detail how are they communicating in chapter 4. To sum this: all pins of RPi is straightly connected to GPIO pins of UniPi through the flat connector. Other building blocks such as relays are connected through MCP23008 expanding chip and communicated via I2C protocol. I2C protocol is operating through 3th and 5th pin of RPi.

I've designed the simulator for bistable and monostable loaded systems based on Python. Also, I've designed simulators for the monostable system based on Node-Red and Mervis. The simulators are working stable and have options to change the parameters of simulations and visualize the process of simulation. For Python-based simulators, it's possible to adjust many parameters such as stroke, diameter, the working pressure of the cylinder, the width of the sensors, load of the cylinder, the mounting angle of the loaded cylinder. Depending on the parameters, which can easily be found from documentation of the specific cylinder, Python-based simulator will calculate the time, run the simulation, visualize it and draw a graph in real-time with the position of the piston and logical values on the first and second switcher. For Node-Red and Mervis based simulators it's necessary to set the time of simulation directly, it will run the simulation and visualize it. Visualization in Node-Red and Mevis based simulators is less sophisticated than in Python-based simulators, but it does its job and shows the state of the simulation. Basically, I've developed a digital twin of a specific system in Python and simpler digital twins in Node-Red and Mervis.

During the work on this bachelor thesis I've looked closely at a lot of interesting technologies like PLC programming, programming of the microcontrollers, pneumatic systems, and automation. Although I've worked only with the simulation of field level elements, I've come to realize, that PLC and pneumatic systems are a very powerful instrument for industrial automation. Because of specific program language and construction of it, PLC is a very efficient automatizing tool.

As regards the programming of microcontrollers: it would be a great instrument for automation because of its size and capabilities, but it hasn't got industrial reliability, so it can't be used fully in industrial automation. For example only in creating a simulation of technological elements or creating "digital twin" of the equipment for testing. Also, it's efficiency depends on software, used with it.

In this bachelor thesis, I used three different software to fulfill the same task: to develop the simulator. And based on the experience of developing the simulator on three different platforms, I must say that Python is the best of three for that. Python is a flexible language with lots of modules and an easy-readable code. Because of that, it's possible to develop a more sophisticated simulator with different modules and capabilities such as visualization, calculation of time inside the script, and plotting. But it would take a much longer time to do that, because of the whole context of OOP and understanding it. For me, it took longer to learn how to make programs in Python, than it took me to learn how to make programs in Mervis, Node-Red.

And as for Node-Red and Mervis. Those are tools mainly applicable for home automation. It has a lot of different predefined functions and modules for many purposes, such as working with sensors, getting the data from it, visualizing the data, and so on. The only problem, that it's har to make something, which is not predefined by creators of those tools. And in my opinion, those are great tools for smart lights or monitoring the temperature in a house or boiler. It would be great to do with it, especially with Node-Red. Mervis is pushing its product in the industrial automation area. But UniPi v1.1 has a lack of industrial reliability, maybe they fixed it in further models because

UniPi v1.1 is a relatively early model.

Python is the most suitable instrument for developing a sophisticated simulation or digital twin, but Mervis and Node-Red are way simpler for home automation or automation in general.

Also developed simulator can be enhanced with some adjustments. Developed feature with plotting is working stably on powerful hardware, but it doesn't work properly on RPi B+ v1.2. The solution here is the optimization of the code or using more powerful hardware. Also for industrial using this kind of simulation, a more sophisticated calculation of the time of the cycle of the pneumatic cylinder would be required.

Bibliography

[1] (2019). Differences between field, control, supervisory, and enterprise levels of automation, [Online]. Available: https://www.syspro.com/blog/erp-for-manufacturing/the-5-layers-of-the-automation-pyramid-and-manufacturing-operations-management/ (visited on 04/20/2020).

[2] (2017). The 5 layers of the automation pyramid and manufacturing operations management, [Online]. Available: https://www.machinedesign.com/automation-iiot/article/21835731/differences-between-field-control-supervisory-and-enterprise-levels-of-automation (visited on 04/20/2020).

[3] (2017). Digital twin, [Online]. Available: https://en.wikipedia.org/wiki/Digital_twin (visited on 07/13/2020).

[4] A. Parr, Hydraulics and pneumatics: a technician's and engineer's guide. 2011, ISBN: 978-0-08-096674-8.

[5] (2019). Pneumatic solenoid valves for pneumatic systems., [Online]. Available: https://tameson.com/solenoid-valves-for-pneumatic-systems.html (visited on 04/13/2020).

[6] (2014). Raspberry pi documentation, [Online]. Available: https://www.raspberrypi.org/documentation/usage/gpio/RsEADME.md (visited on 04/13/2020).

[7] (2014). Gpio port map, [Online]. Available: https://www.raspberrypi.org/documentation/usage/gpio/images/GPIO-Pinout-Diagram-2.png (visited on 04/13/2020).

[8] (2014). Unipi v1.1 page in e-shop, [Online]. Available: https://www.unipi.technology/unipi-1-1-p36 (visited on 04/13/2020).

[9] (2014). Unipi v1.1 technical documentation, [Online]. Available: https://www.unipi.technology/shop/product/download?fileId=986 (visited on 04/13/2020).

[10] B. P. Šimeček, "Řídicí systém inteligentního bytu," p. 20,

[11] (2014). Gpio usage documentation, [Online]. Available: https://www.raspberrypi.org/documentation/usage/gpio/ (visited on 04/13/2020).

[12] (2014). Bcm2835 peripheral specification, [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf (visited on 04/13/2020).

[13] W. Gay, "Physics of the gpio interface," in Exploring the Raspberry Pi 2 with C++, Springer, 2015, pp. 83–94, ISBN: 978-1-4842-1739-9.

[14] (2018). I2c communication protocol, [Online]. Available: https://en.wikipedia.org/wiki/I%C2%B2C#Message_protocols (visited on 04/13/2020).

[15] (2010). Description of iec 61131, [Online]. Available: https://en.wikipedia.org/wiki/IEC_61131 (visited on 06/03/2020).

[16] (2010). Description of iec 61131-3, [Online]. Available: https://en.wikipedia.org/wiki/IEC_61131-3 (visited on 06/03/2020).

[17] K. T. Erickson, "Programmable logic controllers," IEEE potentials, vol. 15, no. 1, pp. 14–17,

[18] Siemens, Tia portal v15.1, version 15. [Online]. Available: https://support.industry.siemens.com/cs/document/109752566/simatic-step-7-and-wincc-v15-trial-download-?dti=0&lc=en-WW.

[19] T. Alves, Openplc editor, version 1.0. [Online]. Available: https://www.openplcproject.com/plcopen-editor/.

[20] (2016). Definition of openplc, [Online]. Available: https://www.openplcproject.com/ (visited on 04/28/2020).

[21] I. E. T. Services, Node red, version 1.0.5. [Online]. Available: https://nodered.org/docs/getting-started/local.

[22] (2017). Definition of nodered, [Online]. Available: https://en.wikipedia.org/wiki/Node-RED (visited on 04/30/2020).

[23] U. technology, Mervis, version 2.1.3.

[24] (2014). Definition of mervis, [Online]. Available: https://www.unipi.technology/products/mervis-43 (visited on 04/30/2020).

[25] (2006). Documentation of smbus module, [Online]. Available: http://wiki.erazor-zone.de/wiki:linux:python:smbus:doc (visited on 05/06/2020).

[26] Y. Magda, Raspberry Pi. Setup and Application Guide. Litres, 2017, ISBN: 978-5-94074-964-6.

[27] Festo, Fluidsim® 5. [Online]. Available: https://www.festo-didactic.com/int-en/learning-systems/software-e-learning/fluidsim/fluidsim-5.htm?fbid=aW50LmVuLjU1Ny4xNy4xOC41OTEuNzk3NQ.

[28] (2016). The core nodes, [Online]. Available: https://nodered.org/docs/user-guide/nodes (visited on 05/18/2020).

[29] (2016). Node red dashboard documentation, [Online]. Available: https://flows.nodered.org/node/node-red-dashboard (visited on 05/18/2020).

[30] (2014). Mervis ide technical documentation, [Online]. Available: https://kb.unipi.technology/en:sw:01-mervis:06-tutorials (visited on 05/18/2020).