IMPLEMENTATION OF A PLC CODE ON A RASPBERRY PI IN CODESYS ENVIRONMENT

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Electrical and Automation Engineering

Valkeakoski campus

Autumn 2019

Anna Makarcheva

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Electrical and Automation Engineering
Valkeakoski

Author Anna Makarcheva **Year** 2019

Subject Implementation of a PLC Code on Raspberry Pi in CODESYS Environment

Supervisor Mika Oinonen

ABSTRACT

This thesis project illustrated a Raspberry Pi microcontroller being used as a replacement for an industrial-grade programmable logic controller (PLC) in a data-collection application. The application, Inspector, is a tool for factory production monitoring provided by a Finnish automation company InSolution Oy, which was the commissioning party of the thesis project. Inspector is comprised of a PLC application collecting real-time data and a web application. The PLC data-collection application normally uses PLCs manufactured by Beckhoff, whereas in this project a program with the same functionality was implemented on a Raspberry Pi in CODESYS programming environment.

The theoretical part of the thesis includes information on the hardware, programming tools, languages and communication protocols used during the empirical part of the project.

In the implementation part of the project, changes were made to the original program to transfer it from TwinCAT to CODESYS environment. Most of the modifications were related to reference libraries, communication to the web client application over TCP/IP and the date and time interface. This part of the project also included the configuration of the inputs and outputs using Raspberry Pi GPIO and Horter I2C input modules.

Finally, the Raspberry Pi program was tested by establishing a connection to a web application, which revealed several issues related to the reliability of the new application in cases of power and network failures. Data persistence was added to the project along with other improvements which eliminated the discovered issues and enhanced the reliability of the application.

As an outcome of the project, a reliable and well-functioning solution was developed for possible use of Raspberry Pi instead of a Beckhoff PLC in the

Inspector tool. The developed solution provided the possibility of a significant increase in the cost-efficiency and flexibility of the tool. It was concluded that a Raspberry Pi can be a feasible replacement for a PLC in certain industrial automation applications. Moreover, the commissioning company decided to continue the project by conducting tests on the new product to prepare it to be released for customer projects.

Keywords Automation, data collection, microcontrollers, PLC programming.

Pages 54 pages including appendices 10 pages.

CONTENTS

Appendices

LIST OF ABBREVIATIONS

ADC	Automated Data Collection
CM	Condition Monitoring
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IO	Input/Output
NOOBS	New Out Of the Box Software
NOVRAM	Non-Volatile Random Access Memory
OS	Operating System
PLC	Programmable Logic Controller
POU	Program Execution Unit
ST	Structured Text
UPS	Uninterruptible Power Supply

1 INTRODUCTION

The commissioning company of the thesis project, InSolution Oy, provides automation solutions for industry and education. One of the company's main products is Inspector – a software tool for production monitoring, that allows users to trace the productivity, availability and efficiency of the machinery at a factory. Inspector collects production data using programmable logical controllers (PLCs) and sends the data to a web application interface. The web application displays numerical information and different figures that allow users to analyse and optimise production processes.

Inspector tool performs data collection using PLCs from Beckhoff, which are programmed in the TwinCAT environment. Beckhoff PLCs provide sufficient reliability and durability for the Inspector application. However, InSolution has faced several difficulties in the implementation of Inspector data-collection application using Beckhoff products. Firstly, all of the used controller models have limitations on the size of the code, which in some cases makes it necessary to reduce the functionality of the program. Expanding the functionality of the application often requires more expensive controller models, which increases the price of the final product. Secondly, the PLC program requires modifications when it is implemented on different controller models, which leads to having several versions of the same program. Therefore, it was decided to look for a solution to replace Beckhoff PLCs in the application. (Katajisto, 2019)

The main idea of the thesis project was to replace a PLC in Inspector with a Raspberry Pi microcontroller. Raspberry Pi can execute PLC code in CODESYS, which is a hardware-independent programming environment. The microcontroller provides sufficient computing power and memory for the Inspector application, and, therefore, would eliminate the issued described above.

Both CODESYS and TwinCAT comply with the IEC 61131-3 standard that includes regulations for PLC programming languages and interfaces. The original PLC code had been written using Structured Text (ST) as the programming language, which can be used in both of these programming environments. Therefore, it is possible to implement the same code on a Beckhoff PLC and Raspberry Pi. However, there are differences in writing and execution of a program, which are related to reference libraries and device hardware. (Codesys, 2019) (TwinCAT, 2019)

However, Beckhoff PLCs are specifically designed to be used in automation and industry, while Raspberry Pi is made mainly for testing and educational purposes. That means that there are significant differences in the hardware design of PLCs and educational microcontrollers, and a

Raspberry cannot fulfil the same requirements as an industrial controller. Another setback of the project is that there is no sufficient information about the limitations and possibilities of Raspberry Pi being used as a PLC and it is difficult to estimate whether it is possible to use a Raspberry Pi for a specific application. The goal of this research project was to check how and to what extent it is feasible to replace a Beckhoff PLC in the Inspector tool.

2 THEORETICAL BACKGROUND

This chapter includes theoretical information on the topics covered in the thesis. Understanding the terms and concepts described in the chapter is essential for the comprehension of the workflow and ideas of the project. The chapter also describes the hardware and software that was utilised in the thesis project, as well as gives information on the used communication protocols.

2.1 PLC programming

The following section provides basic information about what programmable logic controllers are, their cases of use, features, manufacturers and history. It also describes the programming language and environment that were used in the thesis project.

2.1.1 Introduction to programmable logic controllers

A programmable logic controller is a special form of microprocessor-based controller that uses programmable memory to store instructions and to implement functions such as logic, sequencing, timing, counting, and arithmetic in order to control machines and processes. (Bolton, 2015, p. 5)

Figure 1 describes the general idea of a PLC: it executes a program that receives real-time input information, processes it and gives obtained results as output signals.

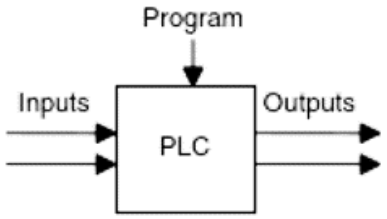

Figure 1. Programmable logic controller (Bolton, 2015, p. 5).

PLC controllers are similar to normal computers, but they are optimized for industrial use, which results in several differences. Firstly, the programming environments of PLCs are simplified to be operated by factory workers and operators who are not necessarily familiar with programming. Therefore, they use graphical programming languages such as Ladder Logic or Function Block Diagram, or simplified textual languages, for instance, Instruction List and Structured Text. These languages use simple logical expressions that can be understood intuitively. Secondly, PLCs are designed to withstand harsh conditions of production environments, such as vibrations, moisture, noise and temperature. Lastly, PLCs have built-in interfacing for inputs and outputs for monitoring, controlling and communicating with other devices. (Bolton, 2015, p. 5), (John & Tiegelkamp, 2010).

The first PLC was invented in 1968 by Dick Morley in response to the request from a US automobile manufacturer GM Hydramatic. The PLC was created as a replacement for old relay logic systems, which needed to be re-wired every time manufacturing process was updated. The first PLC, Modicon 084, was designed to be programmed with ladder logic, which bore resemblance to the relay logic diagrams, in order to be comprehensible for the people used to work with the old systems. Later on, PLC development continued by the introduction of Modbus communication, Manufacturing Automation Protocol, PLC programming PC software, different programming languages and standardisation by International Electrotechnical Commission (IEC). (History of the Programmable Logic Controller (PLC), n.d.).

Nowadays, there is a high variety of PLCs with different characteristics and capabilities available on the market. For instance, the world's best-known PLC manufacturers include ABB, Siemens, Beckhoff, Schneider Electric, Omron, Allen-Bradley.

2.1.2 Structured Text programming language

"So why not just go ahead and use ST all the way? The answer to this is that you can" (Hanssen, 2015, p. 396).

According to John and Tiegelkamp (2010, p.10), classical PLC programming methods that existed earlier, such as Instruction List, Ladder Logic or Control System Function chart had reached their limits. What is more, there were certain differences in both software and hardware of products from different manufacturers, even for the ones, that used the same programming language. These differences were escalating with the development of PLC programming, and, therefore, there was a need for a standardisation of PLC programming, hardware requirements, and manufacturer-independent language concepts. (Hanssen, 2015, p. 20).

International standard IEC 61131 was introduced in 1993 by the IEC and stands for “Industrial-process measurement and control – programmable controllers”. The standard regulates PLC hardware, programming languages and interfaces. The latest version of the standard is IEC 61131-3 which was released in 2013. (Hanssen, 2015, p. 20), (Budimir, 2018).

There are three graphical programming languages and three textual languages defined by the IEC 61131-3 standard. The graphical languages are Ladder Diagram, Function Block Diagram and Sequential Function Chart (graphical version). The textual languages are Instruction List, Structured Text and Sequential Function Chart (textual version). (John & Tiegelkamp, 2010, p. 97).

In this project, ST was used as the main programming language, both in the source Inspector code and in the modified code for Raspberry Pi. ST programs consist of statements divided by semicolons. Statements control the program flow, change the values and manage the calls for program execution units (POUs).

Out of the PC programming languages, ST is comparable to PASCAL and C. Therefore, many programmers who have worked with these languages, find it easy to adapt to the syntax of ST. Structured Text is a High-Level programming language, which means it uses abstract statements describing complex functionality in a very compressed way. An example of the ST code is given in Figure 2. (John & Tiegelkamp, 2010, p. 116).

```
WHILE index1 < 100 DO
    index2 := 0;
    Value := Table1[index1];
    REPEAT
        Table2[index2] := Value + Table2[index2];
        IF Table2[index2] > 32767 THEN
            EXIT;
        END_IF
        index2 := index2 + 1;
    UNTIL index2 >25
    END_REPEAT
    index1 := index1 + 1;
END_WHILE
```

Figure 2. Example of ST code (Hanssen, 2015, p. 395).

John and Tiegelkamp (2010, 116) describe three main advantages of Structured Text when comparing it to Instruction List. Firstly, it provides a very compressed formulation of the programming task. Secondly, there is a clear construction of the program in the statement blocks. Lastly, it has powerful constructs to control the command flow.

On the other hand, these advantages lead to the fact that programs written using ST need to be compiled, and translation to the machine code cannot be accessed directly. Compilation of the code, as well as abstract statements, reduce the efficiency of the code execution. (John & Tiegelkamp, 2010, p. 116).

Moreover, Hanssen states that “ST is first and foremost with arithmetic calculations, processing numbers and in handling structured data types” (Hanssen, 2015). They also mention that when compared to LD, writing code in Structured Text faster and more concise. In addition, some operations cannot be implemented in a graphical programming language. (Hanssen, 2015).

2.1.3 CODESYS

Raspberry Pi was programmed using the CODESYS 3.5 programming tool, developed by Smart Software Solutions GmbH. CODESYS stands for Controller Development System, it is a hardware-independent programming system for industrial automation technology that complies to IEC 61131-3 standard. It supports all five languages described by the standard, as well as other basic PLC programming attributes and features. (Hanssen, 2015, p. 486).

CODESYS is compatible with more than 1 000 device types from over 400 different manufacturers. It is available as a modular single-source runtime system for different device platforms. CODESYS provides multiple add-on components and libraries for it to be used in various types of automation systems. The software is available in complete version for free of charge download on the CODESYS official website. However, some of the additional modules require paid licences. The integration of different editors and add-ons in the CODESYS Development System is illustrated in Figure 3. (Why CODESYS?, 2019).

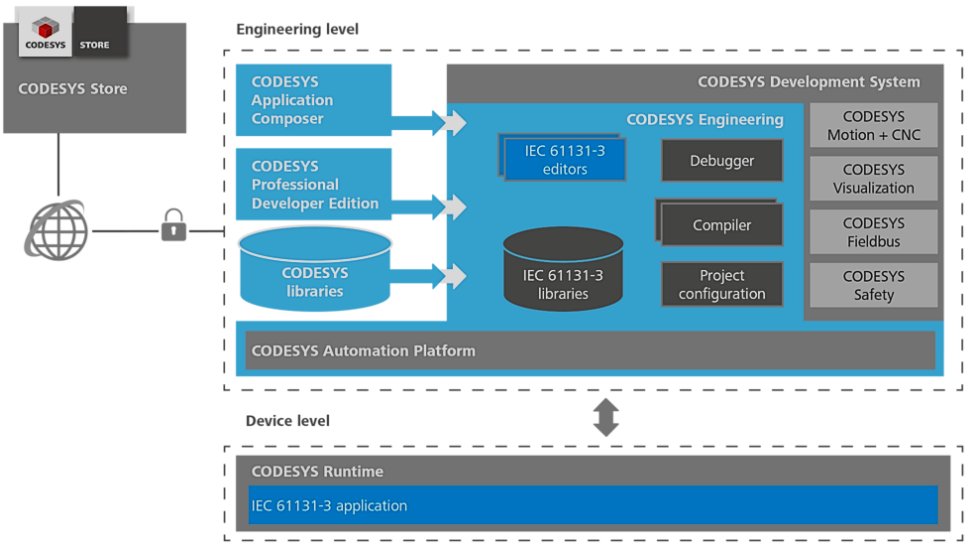

Figure 3. Integration of CODESYS system (Why CODESYS?, 2019).

In the implementation of the thesis, CODESYS runtime system was installed and programmed on a Raspberry PI microcontroller using CODESYS Control for Raspberry Pi SL extension. The package supports multiple communication interfaces of Raspberry Pi, such as I2C, GPIO, One-wire, etc., as well as CODESYS WebVisu visualisation, and several fieldbus protocols. (CODESYS Control for Raspberry Pi SL, 2019).

2.2 **Controller selection and requirements**

This section describes the requirements of the project for the controller in the Inspector application and provides parameters of the currently used controller and its replacement options. The commissioning company suggested using a Raspberry Pi microcontroller. Moreover, two other controller options were considered – Allen-Bradley Micro800 PLC and Arduino microcontroller. The section also provides information on the choice that was made and its reasoning.

2.2.1 Required parameters

The main idea of the project was to develop a solution to replace the Beckhoff PLC in the Inspector application. Normally, Beckhoff BC9020 is used as a controller in the application. Depending on the project specifications, other controller models can be used, such as BC9050 or CX8090, and input/output (IO) cards are added.

The BC9020 Ethernet TCP/IP “Economy plus” Bus Terminal Controller is a Bus Coupler with built-in PLC functionality and a bus interface for Ethernet. It is programmed with TwinCAT software which complies to IEC 61131-3. The image of the controller is given in Figure 4 and its technical specifications are provided in Table 1. (BC9020, 2017).

Figure 4. Beckhoff BC9020 (BC9020, 2017).

Table 1. Beckhoff BC9020 specifications (BC9020, 2017).

Program memory	128 kbytes
Data memory	128 kbytes
Remanent data	2 kbytes
Persistent data	1 kbyte
Runtime system	1 PLC task
PLC cycle time	approx. 1 ms for 1,000 instructions (without I/O cycle, K-bus)
Programming languages	IEC 61131-3 (IL, LD, FBD, SFC, ST)
Programming software	TwinCAT
Ethernet connection	RJ45

BC9020 is suitable for Inspector application with its speed and computing power. However, there are two main setbacks of the implementation with Beckhoff products. Firstly, there are limitations on the program size, which can restrict the functionality of the application from being expanded. Secondly, there is a lack of flexibility when using different controller models. All the Beckhoff controllers are programmed in TwinCAT environment, which has two active versions: TwinCAT 2 and TwinCAT 3. Even though TwinCAT 2 and 3 use the same programming languages and structures, in some cases they require different reference libraries for the same functionality. This leads to having to write several versions of code with the same functionality, which is inconvenient.

The new implementation needs to resolve the issues mentioned above without adding any other restrains to the application. Inspector application requires the controller to be able to perform communication over TCP/IP and have extendable IO.

Several controller models that could fulfil the requirements of the Inspector application have been found. Their parameters are described in the sections below.

2.2.2 Raspberry Pi microcontroller

Raspberry Pi is a low cost, credit-card sized computer which can be connected to a monitor, mouse and keyboard and is capable of performing all the standard functions of a desktop computer with additional possibilities for interaction with the outside world.

This microcontroller was designed by Raspberry Pi educational charity foundation to promote programming in education. The microcontroller is supplied with an affordable price, around USD 35, plentiful of understandable instructions and inspirational ideas, which makes it a good platform to learn programming for people of all ages. Moreover, the manufacturer promotes the idea that learning programming can be seen as entertainment or hobby, both for children and adults. (What is a Raspberry Pi?, n.d.).

The promotional video on the official Raspberry PI website claims: "…we have seen examples of people using the Pi in a variety of amazing interesting projects taking advantage of its size, portability, cost, programmability and connectability" (What is a Raspberry Pi?, n.d.).

Raspberry Pi is a Linux-based computer which has a set of general-purpose input/output (GPIO) pins for controlling electronic components and interaction with the Internet of Things, as illustrated in Figure 5. Even though Raspberry Pi is designed mainly for developing practical skills in programming and building hardware, it is also used in home automation and industrial applications. (Raspberry Pi, 2019).

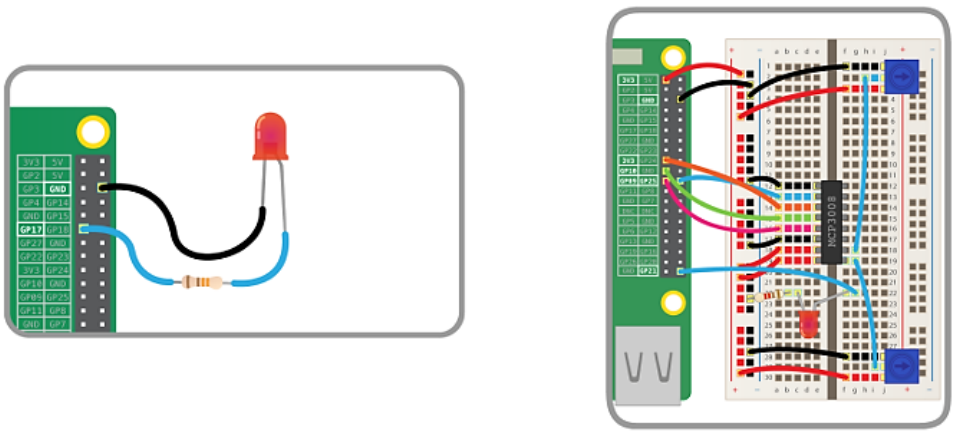

Figure 5. Raspberry Pi GPIO pins (Raspberry Pi, 2019).

The commissioning company provided Raspberry Pi 3 Model B v1.2 for the thesis project. This model, released in 2016, is the earliest model of the third generation of Raspberry Pi. Table 2 provides the specifications of the microcontroller.

Table 2. Specifications of Raspberry Pi 3 model B. (Raspberry Pi 3 Model B, n.d.)

Processor	Quad-Core 1.2GHz Broadcom BCM2837 64bit
RAM	1GB
Wireless connections	BCM43438 wireless LAN and Bluetooth Low Energy (BLE)
Ethernet connection	100 Base Ethernet
GPIO	40 pins, extended
USB connections	4 ports
Media outputs	4 Pole stereo output and composite video port
Screen output	Full-size HDMI
Port for SD card	Micro SD port for loading your operating system and storing data
Power supply	Micro USB power source up to 2.5A
Other ports	CSI camera port for connecting a Raspberry Pi camera; DSI display port for connecting a Raspberry Pi touchscreen display

2.2.3 Allen-Bradley Micro800 programmable logic controllers

The Micro800 PLCs from Allen-Bradley are designed for low-cost standalone automation applications. The controllers of the series are shown in Figure 6. All the models in the series are programmed in the same programming environment which supports the following IEC61131-3 programming languages: Ladder Diagram, Function Block Diagram and Structured Text. The models Micro820, Micro850 and Micro870 have embedded Ethernet ports and support communication over TCP/IP. Micro820 PLC has similar specifications to the Beckhoff BC9020, whereas Models 850 and 870 provide more IO points, program memory and computing power. Allen-Bradley Micro800 PLCs already have embedded IO, size of which varies for different models, unlike the Beckhoff controller, that requires additional IO cards. Thus, different controller models of Micro800 series can be used depending on the project requirements, providing the application with flexibility. (Micro800 Programmable Controller Family Selection Guide, 2019).

Figure 6. Allen-Bradley Micro800 controllers (Micro800 Programmable Controller Family Selection Guide, 2019).

2.2.4 Arduino microcontroller

Arduino is an open-source platform, which contains a microcontroller and an integrated development environment software for programming the controller from a computer. Arduino can be used in various electronics applications by students, teachers, programmers, artists and scientist. Arduino provides an integrated development environment (IDE) software that allows to connect and program to the microcontroller via USB from a computer. Arduino IDE uses C/C++ programming languages. However, there also exists other software for programming Arduino boards using different programming languages. For example, Visual Studio provides an IDE specifically for Arduino. Arduino can also be programmed using IEC 61311-3 languages with logi.CAD 3 software. (What is Arduino?, 2019)

Arduino UNO WIFI Rev2 was chosen for this project. This board has an embedded Wi-Fi module that provides the possibility of communication over TCP/IP for the Inspector application. The technical specifications of the selected board are given in Table 3. (Arduino UNO WiFi Rev2, 2019)

Table 3. Specifications of Arduino UNO WIFI Rev2 board (Arduino UNO WiFi Rev2, 2019).

Microcontroller	ATMEGA4809
Operating Voltage	5V
Input Voltage (recommended)	7 - 12V

Input Voltage (limit)	6 - 12V
Digital I/O Pins	14 — 5 Provide PWM Output
PWM Digital I/O Pins	5
Analogue Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	48 KB (ATMEGA4809)
SRAM	6,144 B (ATMEGA4809)
EEPROM	256 Bytes (ATMEGA4809)
Clock Speed	16 MHz
LED_BUILTIN	25
Length	68.6 mm
Width	53.4 mm
Weight	25 g

2.2.5 Selection

After comparing the parameters of the controllers described above, it was decided to use a Raspberry Pi. Allen-Bradley PLCs could improve more program memory and flexibility for the Inspector application, but they would not improve its cost-efficiency. Microcontrollers have significantly lower prices than industrial PLCs from Beckhoff and Allen-Bradley. Raspberry Pi has more IO pins, RAM and flash memory and more powerful processor, than Arduino. It also does not have any limitations for program size specified. Having considered all the facts stated above, it was decided by the author of the thesis and the commissioning company to use Raspberry Pi microcontroller for this project.

2.3 **Inspector software tool**

This section provides information on Inspector software, part of which will be transferred from Beckhoff PLC to be implemented on a Raspberry Pi. It provides a general description of the functionality and usage of the tool. In addition, it gives technical information about the implementation of the tool and its features used in this thesis project.

2.3.1 Functional description of Inspector software

The increase in production profitability is the main target of the Inspector software tool. The tool collects information about the production losses and their reasons, resources utilization and other performance indicators. Based on the collected data, Inspector gives an overview of the production processes. A visual representation of performance indicators provided by Inspector is shown in Figure 7.

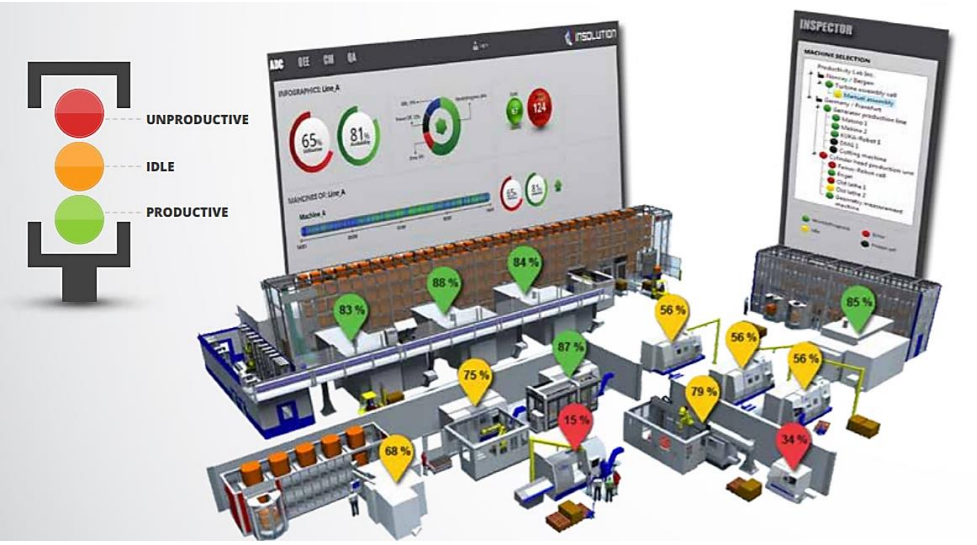

Figure 7. Inspector software in production (Inspector Production monitoring).

Inspector does not only show the production losses but also helps to identify and analyse their causes. The chart in Figure 8 illustrates how losses can cut down the production time. There are six main losses in the production: stops, setup and adjustment, idle time, reduced speed, quality errors and remachining and startup errors. These losses affect the three factors of overall equipment effectiveness (OEE): availability, speed and quality. The chart shows how the theoretical production time of 8760 hours per year is reduced to actual production time, which is always smaller than the theoretical, due to planned stops and the six big losses.

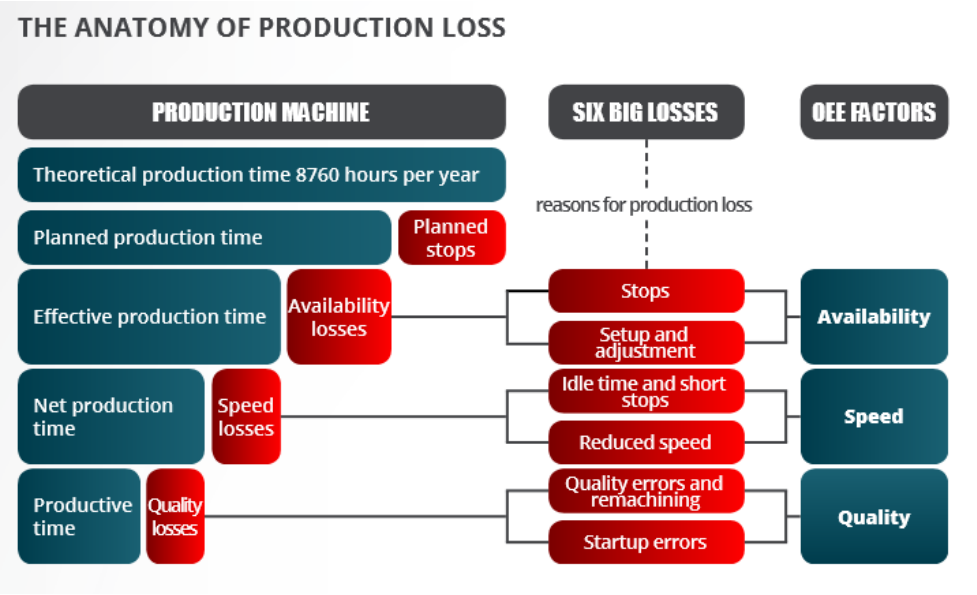

Figure 8. Anatomy of production loss (Inspector Production monitoring).

What is more, Inspector can perform quality monitoring and resource organising. It can be integrated with existing factory systems such as MES

(Manufacturing Execution Systems), Maintenance and ERP (Enterprise Resource Planning). The software provides a modular system, which can be adjusted to the needs of a particular customer. (Inspector Production monitoring).

2.3.2 Technical information on Inspector software

Inspector software uses automated data collection (ADC) to gather real-time information about production states. Production data is acquired by PLCs. The application uses PLCs manufactured by Beckhoff which are programmed using TwinCAT 2 or TwinCAT 3 software. Depending on the requirements of a specific project, different controller models are used. Most used PLC models are BC9020, BC9050 and CX8090. In some cases, the PLC is placed in the Inspector Box – a box containing all the hardware components and wiring necessary for PLC setup. Inspector Box also has three status LEDs and nine buttons for manual inputs, which are controlled by the PLC.

PLCs collect data from machine signals and sensors and send real-time information about machine state to the Inspector web application. The web application checks whether the state is productive, idle or unproductive and logs the information to the database. In the graphical user interface (GUI) of the application, the states can be configured, which means the input signals, name, meaning and colour of a state can be set.

The users of Inspector can give reason codes to the unproductive and idle states. A reason code gives a short explanation of why the production has been stopped, for instance, “break”, “maintenance”, “emergency”, etc. The reason codes can be added manually using GUI or by pressing a button of a reason code box.

Inspector software also provides condition monitoring (CM) module for tracking analogue input data. Collected data can be retraced from graphs drawn in the GUI or be “transferred” to ADC data, which means that machine states can be read based on analogue input values. CM data can be scaled and have its sample rate set in a way that would provide most informative and simplified image of the production.

In the Inspector application, PLCs communicate with the web application over TCP/IP protocol, which is described more in detail in the next chapter.

2.4 **Communication protocols**

Communication protocols are formal descriptions of digital message formats and rules, required to exchange messages in or between computing systems. The protocols describe communication by defining the rules its of authentication, error handling, signalling, syntax, semantics

and synchronisation. (Communicatoin Protocol, 2019). The section below provides descriptions of the communication protocols used in this project.

2.4.1 TCP/IP

TCP/IP, or the Transmission Control Protocol/Internet Protocol, is a family of network protocols that are used for connection and communication of devices in the Internet or private networks. TCP/IP regulates data exchange by specifying how it can be divided into packets, addressed, transmitted, routed and received at the destination. TCP/IP protocols were developed to provide network reliability and sustainability. (Rouse, 2019)

TCP/IP group has two main protocols. TCP is responsible for the assembly of the data into packets before transmission and reading of those packets on the receiving devices. IP describes the addressing and routing of data packets to ensure that they reach the correct destination. Gateway computers use IP addresses to find where to forward the data. (Rouse, 2019)

TCP/IP protocol uses the client/server model, in which the server computer provides service to another machine or user. TCP/IP communication happens at several different levels, and it can be divided into four layers. *The physical layer* contains protocols that operate on link and are responsible for physical connections inside the network, such as Ethernet or Address Resolution Protocol. *The network layer,* sometimes referred as the Internet level, is related to interconnection of independent networks, data packets and their transmission across the networks. The network layer includes IP and Control Message Protocol, the protocol for reporting of errors. *The transport layer* maintains end-to-end communication between hosts and provides flow control, multiplexing and reliability. The protocols of the transport layer are TCP and User Datagram Protocol, which can sometimes replace TCP. *The application layer* is needed for standardised data exchange between applications. It consists of such protocols as Hypertext Transfer Protocol, File Transfer Protocol, Post Office Protocol 3, Simple Mail Transfer Protocol and Simple Network Management Protocol. (Rouse, 2019).

The TCP/IP protocols are stateless, i.e., each request is unrelated to the previous one and is considered new. Being stateless allows the network paths to be used continuously. However, the transport layer is stateful, because it needs to transfer a message in packets and keep the connection alive until the whole message is received and reassembled. (Rouse, 2019).

2.4.2 Secure Shell (SSH) protocol

Secure Shell is a communication protocol that provides a secure login to a remote computer. SSH uses the client/server model, where the connection

to the server computer is initiated by the client. The SSH client operates the setup of the connection and uses public-key cryptography to verify the identity of the SSH server. The simplified model of an SSH connection is illustrated in Figure 9. (Ylonen, 1996, pp. 37-42).

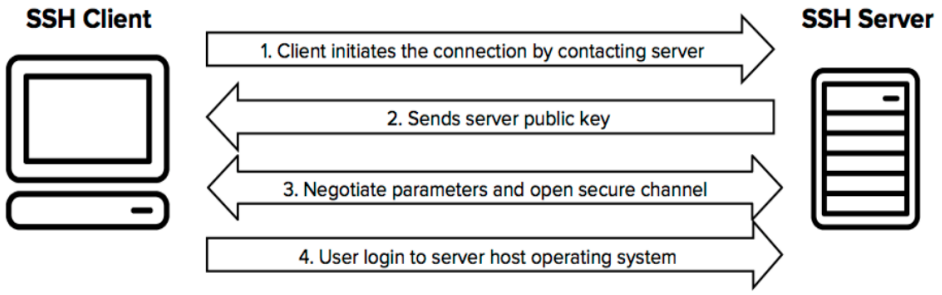

Figure 9. SSH connection scheme (SSH (Secure Shell), 2018).

The most common uses of SSH include providing secure access for users and automated processes, interactive and automated file transfers, issuing remote commands and managing network infrastructure and other mission-critical system components (Ylonen, 1996, pp. 37-42).

The SSH protocol was invented in 1995 by Tatu Ylonen because of a hacking event in the Finnish university network. A password sniffer had been installed to a server that was connected to the backbone, it had stolen thousands of usernames and passwords before it was noticed. Ylonen's company was among the ones that were affected by the incident, and he developed a solution which would help him perform a secure remote login over the Internet. Later on, the protocol was improved and standardised. Nowadays Secure Shell protocol is utilised for managing more than half of web servers in the world and almost every Linux or Unix computer. (SSH (Secure Shell), 2018).

2.4.3 I2C protocol

Inter-Integrated Circuit (I2C), sometimes also referred as Inter-IC, IIC or I²C, is a widely used serial bus protocol designed by Philips in the early 1980s. The protocol is used for communication between electrical components on the same board, low-speed devices, such as microcontrollers, IO modules or other peripherals in the embedded systems. (I2C – What's That?, n.d.), (I2C Info – I2C Bus, Interface and Protocol, 2019).

The protocol is flexible and easy to use, as it is based on simple master-slave relationships between the components and requires only two wires to connect an almost unlimited number of controllers. The bus lines are SDA (serial data) and SCL (serial clock). Both of them require pull-up resistors to the positive supply voltage. Since the bus clock is generated by the master device, I2C bus communication does not have strict

specifications for baud rate as, for instance, RS232 protocol. What is more, I2C bus is a true multi-master that provides collision detection and arbitration. An example of an I2C bus connection is shown in Figure 10. (I2C – What's That?, n.d.), (I2C Info – I2C Bus, Interface and Protocol, 2019).

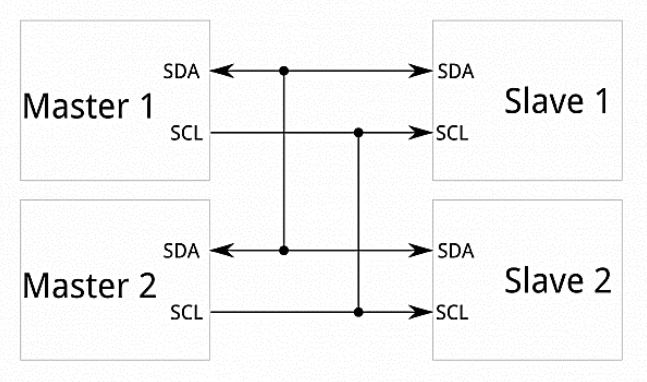

Figure 10. Example of I2C bus connection (I2C, n.d.).

The clock frequency in I2C is 100kHz, which means that the speed of communication is 100kbit/s. However, there also exists 400kHz Fast mode, 3.4 MHz High-speed mode and 5 MHz Ultra-fast mode. (I2C Info – I2C Bus, Interface and Protocol, 2019)

Communication over the I2C protocol happens by transferring 8-bit messages. All the slave devices on I2C bus need to have unique 7-bit addresses by which masters can identify the devices on the bus. Data is transferred over SDA and the message frames are regulated by SCL. The scheme of a message transferred over SDA and SCA lines is shown in Figure 11, where S is the start condition, P – stop condition and B1, B2 .. Bn are the bits of data. (I2C Info – I2C Bus, Interface and Protocol, 2019)

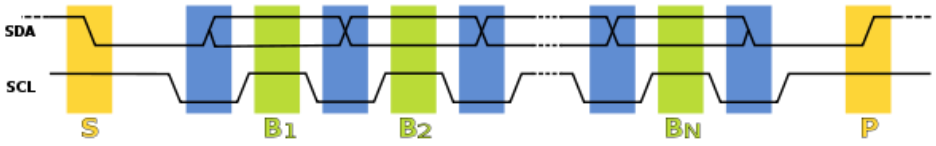

Figure 11. Message transfer over I2C protocol (I2C Info – I2C Bus, Interface and Protocol, 2019).

3 IMPLEMENTATION

This chapter illustrates the practical part of the project. It describes the actions that were taken in order to achieve the goals of the project. The implementation part of the thesis includes software installations on Raspberry Pi and the PC, exporting the program from the TwinCAT environment to CODESYS and adjusting it to work on the microcontroller.

3.1 **Setup preparations**

Several preparations needed to be done on the computer and Raspberry Pi before the Inspector code could be executed on Raspberry Pi. Firstly, the basic settings and installations were performed on Raspberry Pi. Secondly, installations on the computer side were performed. Finally, the Inspector PLC program was exported from the TwinCAT environment and imported to CODESYS.

At the beginning, hardware prototyping was done on Raspberry Model B Revision 2.0. This model is older and has lower hardware specifications than Raspberry Pi 3 Model B v1.2 used in the final setup. However, both models support the CODESYS Control Module and can provide all the needed functionality. The newer model was used in the final setup, where it was placed in the Inspector Box and connected to the IO modules and the reason code buttons.

3.1.1 Preparation of Raspberry PI

First of all, an operating system (OS) had to be installed on the Raspberry Pi. NOOBS, which stands for New Out Of the Box Software, is a simple OS installer, provided by Raspberry Pi Foundation. It can be downloaded from the official Raspberry Pi website and extracted to an empty SD card. Once the card with NOOBS is inserted to a Raspberry Pi, it prompts a selection of operating systems that can be installed on the Raspberry Pi. The installation of an operational system with NOOBS can be performed online or offline, depending on the desired OS and NOOBS version.

The microcontroller was connected to the office network using Ethernet cable, which also provided the Internet connection for the Raspberry Pi. The Raspberry Pi also had a keyboard and a mouse connected to it via standard USB, and an HDMI connection to a monitor. The SD card with NOOBS v3.2.1. was inserted into the microcontroller. Finally, Raspberry Pi was powered by a mini-USB 5V supply. The Raspberry Pi with these connections is pictured in Figure 12.

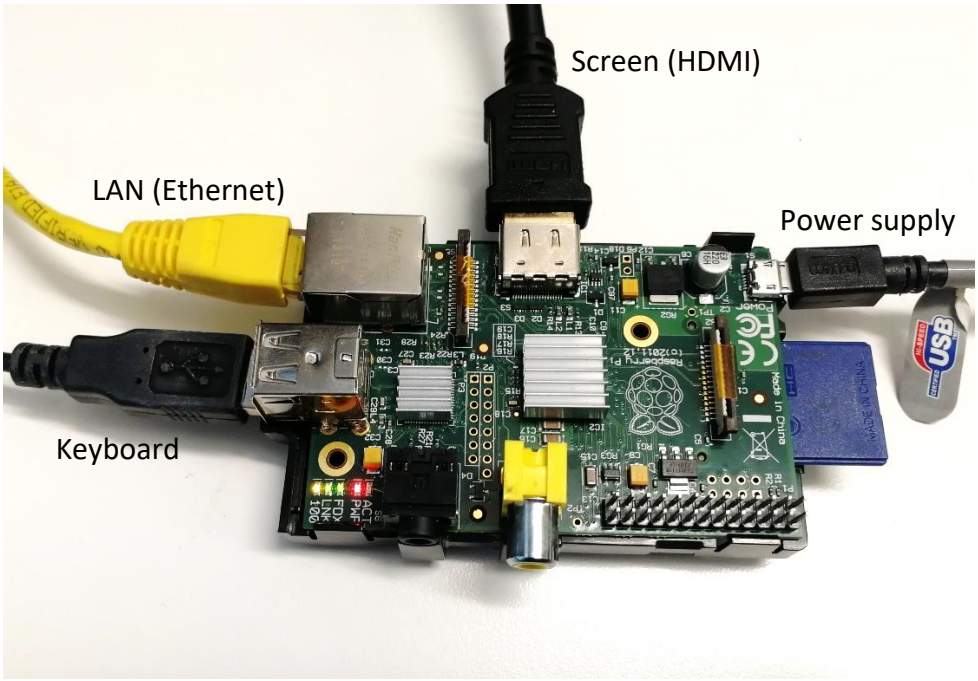

Figure 12. Raspberry Pi connections.

When the Raspberry Pi was turned on, the OS selection was made in NOOBS – Raspbian, the desktop version without additional software. Raspbian is the official supported Linux-based OS for Raspberry Pi. More information about the OS version can be found from the screenshot in Figure 13.

```
pi@raspberrypi:~ $ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 10 (buster)"
NAME="Raspbian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org/"
```

Figure 13. Raspberry Pi OS version.

Once OS installation was completed, some of the basic settings needed to be checked. Firstly, the localisation of the Raspberry Pi was set to Helsinki, Finland. Since Raspberry Pi does not have real-time clock, it synchronizes its time settings with the Internet. Localisation settings configure it to use the right time zone. Secondly, the interfacing values were checked and SSH and I2C communication interfaces were enabled, as shown in Figure 14.

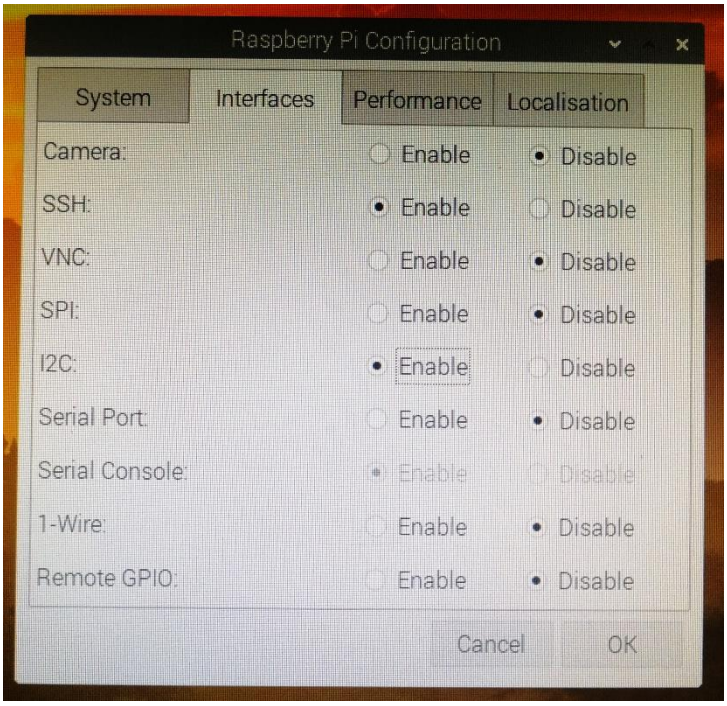

Figure 14. Configuration of the communication interfaces of Raspberry Pi.

Lastly, it was necessary to find the IP address of the Raspberry Pi so that the microcontroller could be accessed and controlled from the computer, connected to the same network. Command "*ifconfig*" was used to check the parameters of the network interfaces on the Raspberry Pi. Figure 15 shows the information that was returned by this command, including the IP address of the device: 192.168.1.181.

```
pi@raspberrypi:~ $ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.181  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::c00e:a0b7:a1be:dac9  prefixlen 64  scopeid 0x20<link>
        ether b8:27:eb:69:60:c6  txqueuelen 1000  (Ethernet)
        RX packets 80687  bytes 108002860 (102.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18227  bytes 5592134 (5.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 138  bytes 6676 (6.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 138  bytes 6676 (6.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlan0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether b8:27:eb:3c:35:93  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 15. Network interfaces of the Raspberry Pi.

After the steps described above had been completed, the Raspberry Pi was prepared and accessible for further installations, which were performed from the computer side.

3.1.2 Installations on computer

On the computer side CODESYS software was installed, version 3.5 SP14 Patch 2. In addition to that, "CODESYS Control for Raspberry Pi SL" module was downloaded from official CODESYS store and installed. The functionality of this package allows to connect to the Raspberry Pi from PC and install CODESYS runtime on it. With this package installed, the PLC code written in the software on the computer can be downloaded to the microcontroller and executed, the same way as on common PLCs.

In order to allow CODESYS software to execute commands and download data to the Raspberry Pi, SSH connection needed to be established between the computer and the microcontroller. For this purpose, an open-source software named PuTTY was used. Version 0.72 for Windows x64, released in July 2019 was downloaded and installed to the computer. The screenshots in Figure 16 and Figure 17 illustrate establishing a connection and log in to the Raspberry Pi using PuTTY.

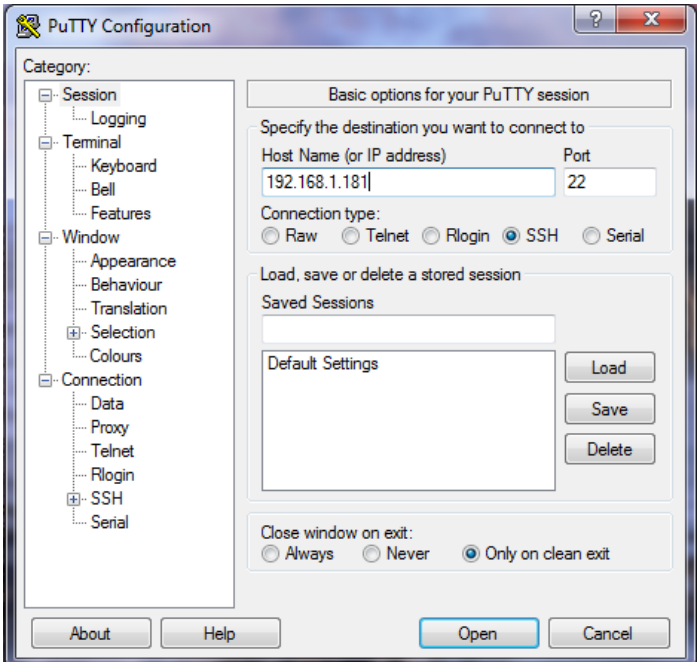

Figure 16. Establishing an SSH connection to Raspberry Pi using PuTTY.

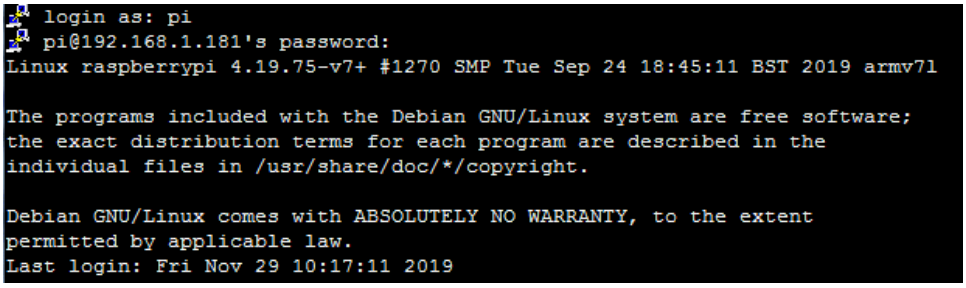

Figure 17. Log in to Raspberry Pi via SSH using PuTTY.

After the SSH connection was established, the Raspberry Pi could be accessed and controlled by the CODESYS computer software. CODESYS runtime was installed on the Raspberry Pi directly from the computer application. Figure 18 shows the interface for installation and control of CODESYS runtime on Raspberry Pi.

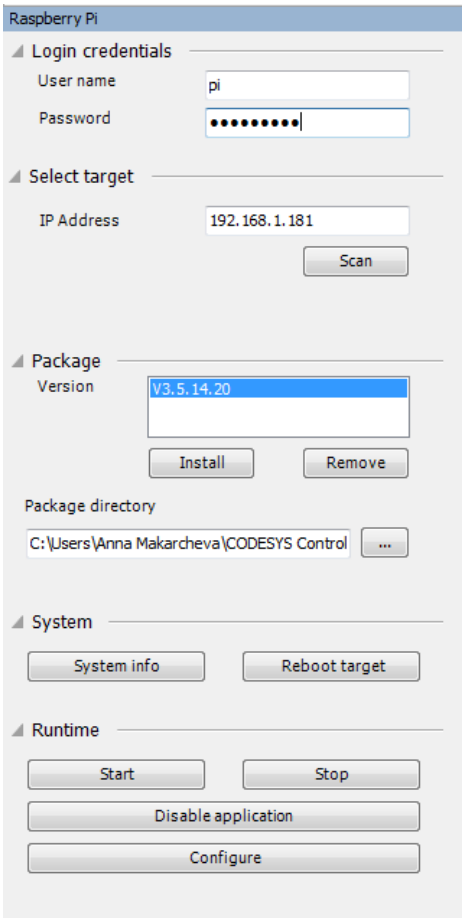

Figure 18. Control of Raspberry Pi runtime in CODESYS.

When the CODESYS Control runtime system had been installed on the Raspberry Pi, it was possible to log in to the runtime, download a program and execute it. Figure 19 shows the connection to the Raspberry Pi as a target device.

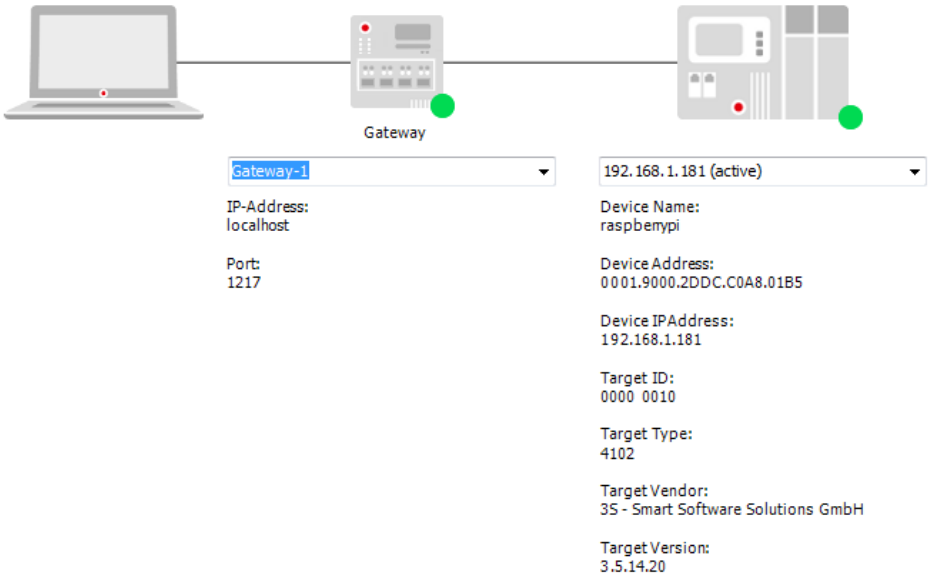

Figure 19. Connection to Raspberry Pi from CODESYS.

3.1.3 Exporting Inspector code from TwinCAT to CODESYS

InSolution Oy has developed several versions of Inspector program for different PLC types, which correspond to different project requirements. In this project, it was decided to use the version written for TwinCAT 3 environment for CX8090. Unlike TwinCAT 2, TwinCAT 3 has the functionality to export files in the PLCopenXml format, which allows the whole code to be imported directly to CODESYS.

After the code had been imported in CODESYS, the whole PLC project, including all the programs, functions and function blocks, as well as global variables and custom data types were present in the new project. The new program was named *Inspector RPi*.

3.2 **Adjusting Inspector code for Raspberry Pi**

The imported Inspector program could not be built in the CODESYS environment due to almost two hundred compilation errors, as shown in the screenshot in Figure 20. Most of the errors were caused by functions or function blocks not being defined. The reason for it was that the TwinCAT libraries used in the original code were not imported to CODESYS and therefore functionality defined in those libraries was not available in the new code.

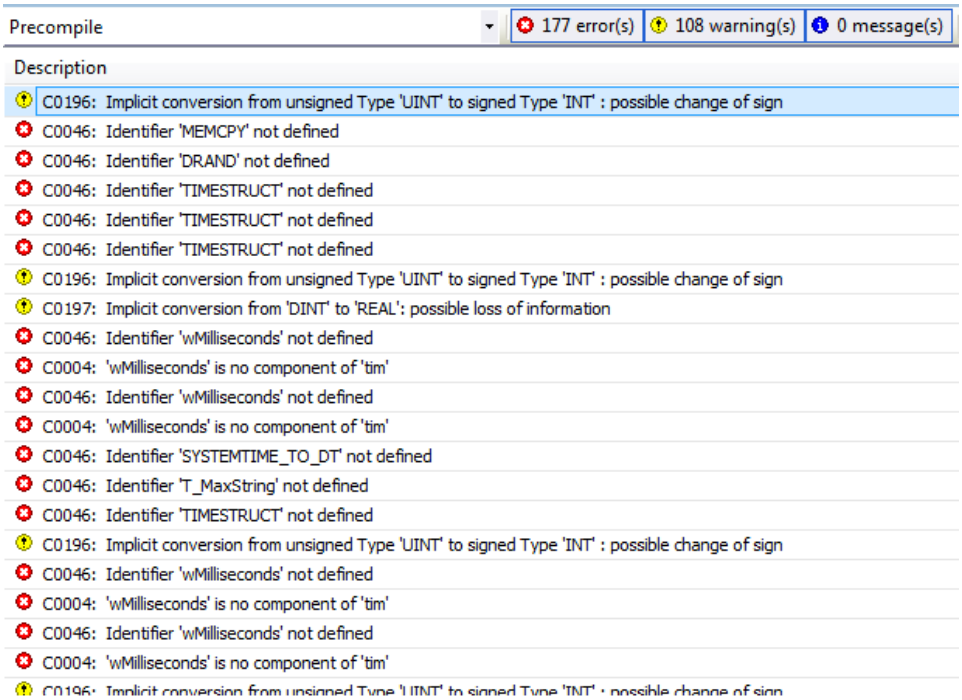

Figure 20. Compilation errors after importing TwinCAT program to CODESYS environment.

The first thing that was tried in order to fix the references, was to add the TwinCAT libraries to the CODESYS project. It was possible as one of the TwinCAT library extensions, ".lib", is also supported by CODESYS. However, when the library files were added, there were still two compilation errors. The errors were inside the "Standard" library, which meant that the library was not supported by the CODESYS environment. Other TwinCAT libraries in the project were dependent on the "Standard" library and each other. Therefore, the TwinCAT libraries could not be used in the CODESYS project and it was reasonable to use the equivalent libraries from CODESYS official store.

3.2.1 Replacing libraries

First of all, CODESYS libraries that provide functionality that is equivalent to the TwinCAT libraries used in the initial application needed to be found. Table 4 illustrates the tables that were used in the old Inspector PLC program and their versions.

Table 4. TwinCAT libraries.

Name	Effective version
Tc2_Standard	3.3.2.0
Tc2_System	3.4.18.0
Tc2_TcpIp	3.3.3.0
Tc2_Utilities	3.3.27.0
Tc3_Module	3.3.18.0

Every function and function block that caused a compilation error needed to be found in a CODESYS library and then the library was added Inspector RPi project. If the names of the functions/function blocks were different from the old ones, they were changed using the “Find and Replace” function. In some cases, new functions/function blocks required different arguments and data types, so they were modified manually.

The search for libraries was performed in CODESYS official documentation, store and Library Manager. There was no information on the usage of the libraries available other than official documentation, which in some cases was insufficient. Therefore, different libraries had been tested, until suitable ones were found.

The changes to the Inspector RPi were done so that the project could be compiled with the least possible changes to its original implementation. Table 5 contains the list of libraries that were added to the final version of the CODESYS project.

Table 5. CODESYS libraries.

Name	Effective version
CmpErrors	3.3.1.40
CommonPacketFormat Interfaces	3.5.6.0
MemoryUtils	3.3.13.0
Standard	3.5.14.0
SysSocket	3.5.14.0
Time and Date	3.5.7.0
Util	3.5.14.0
CAA Memory	3.5.12.0
IODrvEthernet	3.5.14.0

It can be seen from Table 4 and Table 5 that TwinCAT and CODESYS libraries used in the PLC and Raspberry Pi programs are completely different in both names and functionality. Also, the CODESYS project required a bigger number of libraries than the original project. Due to these significant differences in the dependences of the programs, it was challenging to find suitable libraries for Inspector RPi.

Moreover, some of the errors could not be fixed by adding libraries and changing names of the functions/function blocks, which meant that some parts of the program would have to be rewritten in a new way. This problem concerned getting the current time on the controller and TCP/IP communication. The process of rewriting these sections is described in detail in the sections 3.2.2 and 3.2.3.

3.2.2 Rewriting date and time functions

Getting the current time of the controller is important for the Inspector tool because it needs to register the accurate time when changes in a production process happen. It allows the tool to retrace the process and perform calculations concerning, for example, durations of productive/non-productive states.

Unlike Beckhoff PLCs, Raspberry Pi microcontrollers do not have real-time clock and synchronise their current time with the Internet. Also, data types related to date and time and their handling are different in CODESYS and TwinCAT environments. Therefore, it was necessary to find some functionality in CODESYS libraries that would provide the program with the current date and time of the controller.

Several libraries and functions for retrieving the current time from the controller are available in CODESYS environment. Not all of them are suitable for Raspberry Pi, but after testing a few different approaches, a function block that returns current time on the Raspberry Pi was created and called *GetDT.*

GetDT is based on DTU library. A function block called *GetDateAndTime* from this library returns UTC (Coordinated Universal Time). To get local time, time zone information is set during the initialisation of *GetDT* using *SetTimeZoneInformation* function block. Moreover, information about Daylight Saving Time is checked from *GetDateAndTime* before returning the result. During the summer period, one hour is added to the result. As a result, *GetDT* returns current local time with Daylight Saving Time taken into account. The full code of the function block can be found in Appendix 1.

3.2.3 Rewriting TCP/IP communication

Communication of the controller with the Inspector web application is essential for the data collection application, as the web application filters and saves the collected data to the database for future retracing and analysis. Thus, all the data acquired by the controller needs to be sent to the web application server. The part of the program responsible for TCP/IP communication had to be rewritten completely, when the program was transferred from TwinCAT to CODESYS environment, due to significant changes in hardware and libraries.

The main difference between the new implementation and the old communication logic was related to the control of the network sockets. In the old code, TCP socket is opened and closed on each transmission, whereas in the new implementation the connection is established during initialisation of the program and the socket is never closed by the client. Other than that, the new function blocks follow the logic of the old

program and used mostly the same variables. The POUs were composed in such way that they could easily be integrated into the program, without requiring changes in the other parts of the code.

The section of the program responsible for TCP/IP communication consists of three function blocks: *CONNECT*, *SEND* and *RECEIVE*. Their communication functionality is based on *SysSocketCom* library. All three function blocks are called constantly as soon as the program is initialised. Each of them is comprised of several steps. The function blocks communicate with each other and with the rest of the program via shared and global variables, so that the steps inside them are changed accordingly. They also update the information about the communication status and errors on the global program level.

Each of the function blocks is described more in detail and illustrated schematically below. The full text of the of *CONNECT, SEND* and *RECEIVE* function blocks written for Inspector RPi can be found in Appendices 2, 3, and 4 respectively.

First function block, *CONNECT,* is based on *SysSockCreate* and *SysSockConnect* functions. In the step 0 connection parameters are initialised and a new socket is created on the controller. As soon as everything is ready for connection, the function block proceeds to step 10 where the controller attempts to connect to the socket. If the connection is successful, the step changes to 11, which is as simple as microcontroller staying in the connected state. If the attempt to connect fails in step 10 or the connection gets lost in step 11, the function block goes to the error state, step 999. In this step, the code of the error gets logged so that the problem can be found and analysed afterwards. After the error is recorded, the program returns to step 0 and tries to establish the connection again. The graphical representation of the logic of the *CONNECT* function block can be found in Figure 21.

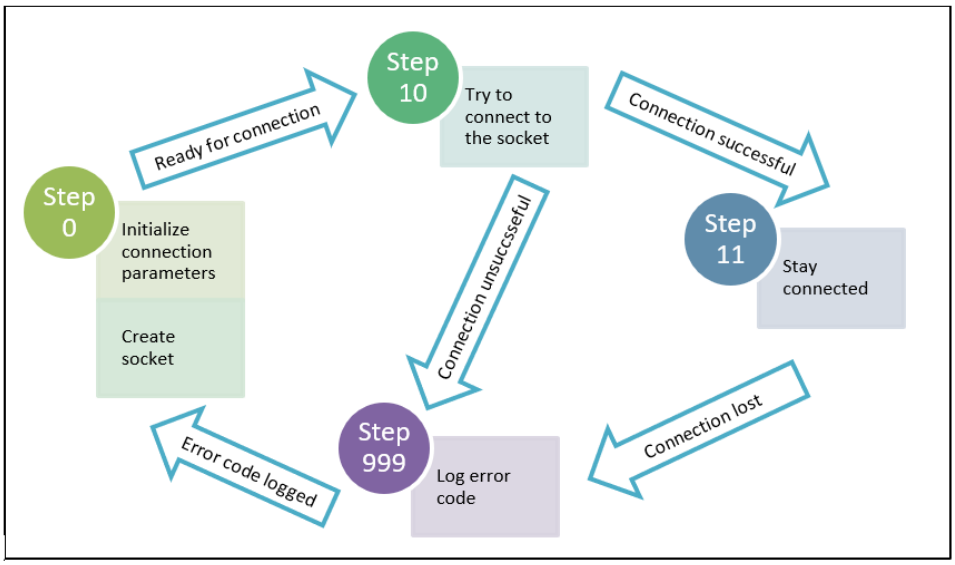

Figure 21. Scheme of the *CONNECT* function block.

Second function block, *SEND*, begins with Step 0, in which it is waiting until the program has data to send. When the function block gets the data, it checks that there is an active connection to the server and the controller is not receiving data over TCP at the moment. As soon as both of these conditions are true, the function block goes to step 11. In this step, the data is sent by *SysSockSend* function, and the send counter is incremented to signify that the data has been sent. After that, the program returns to step 0. However, if an error or a timeout happens during sending, the function block goes to step 999 to log the error, and then proceeds to step 0. The graphical representation of the logic of the *SEND* function block can be found from Figure 22.

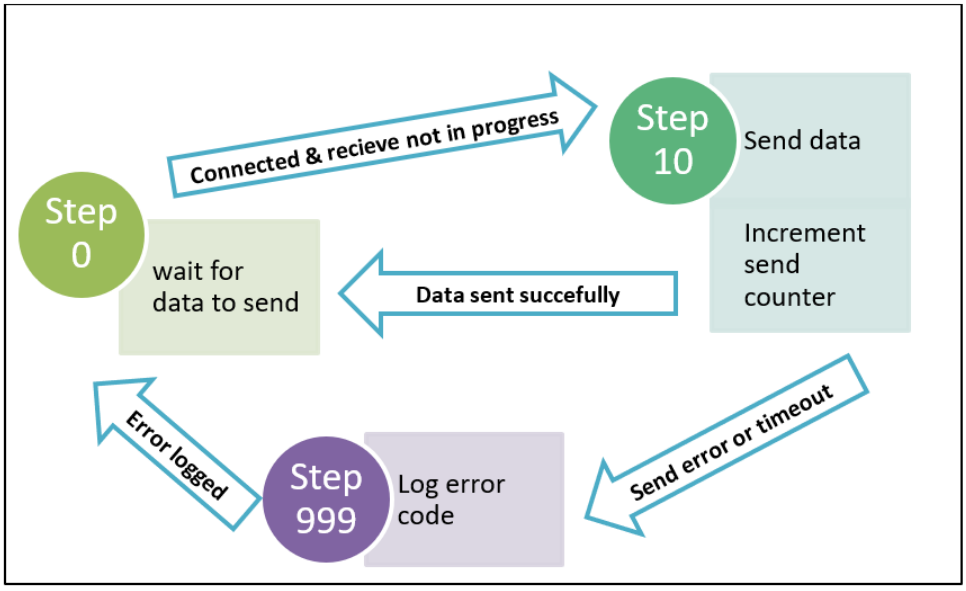

Figure 22. Scheme of the *SEND* function block.

Third function clock, *RECEIVE*, also begins with step 0. In this step receive buffer is initialised. If the controller is connected but is not sending data to the server, the function block proceeds to step 10, where the data is received. In this step, *SysSockRecv* function is used. If the data is obtained successfully, it is parsed and saved to buffer. If the receiving is not successful, then the error gets logged in step 999. In the end, the function block returns to step 0. The graphical representation of the logic of the *RECIEVE* function block can be found from Figure 23.

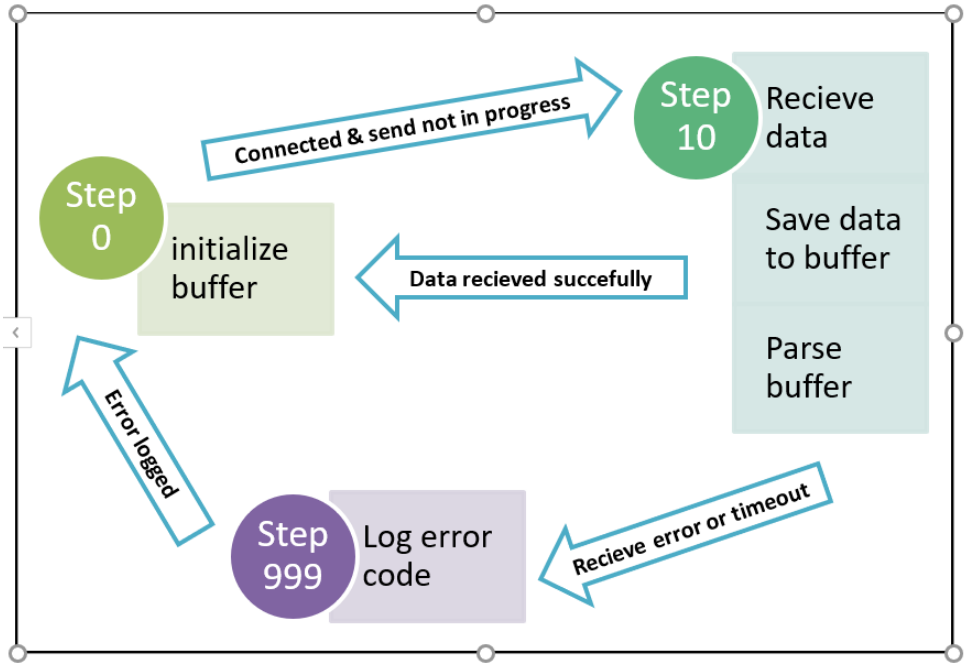

Figure 23. Scheme of the *RECIEVE* function block.

3.3 IO setup

Inspector RPi uses analogue and digital inputs to collect data from the monitored devices. Also, the Inspector Box has nine reason code buttons which are configured as digital inputs for the controller and three status LEDs as digital outputs.

17 out of 40 GPIO pins of the Raspberry Pi B can be used as digital inputs or outputs in the CODESYS program. In Inspector RPi these pins were connected to buttons and LEDs of the Inspector Box. The IO of the Raspberry Pi was extended by adding digital and analogue input modules from Horter. The modules were used to collect information from external devices. Configuration of the IO is described more in detail in the sections below.

3.3.1 GPIO of Raspberry Pi

The interface for the control of Raspberry PI GPIO allows the pins to be configured as inputs or outputs. It also shows the memory addresses where the IO values are stored and provides the possibility to link variables to them. The variables for buttons and LEDs were linked to corresponding pins via memory addresses as shown in the screenshot in Figure 24.

```
(*Digital GPIO outputs*)
output1         AT %QX2.5:          BOOL;
output2         AT %QX3.2:          BOOL;
output3         AT %QX2.4:          BOOL;
(*Digital GPIO inputs*)
input1          AT %IX14.3:         BOOL;
input2          AT %IX12.6:         BOOL;
input3          AT %IX15.1:         BOOL;
input4          AT %IX14.0:         BOOL;
input5          AT %IX13.4:         BOOL;
input6          AT %IX14.6:         BOOL;
input7          AT %IX13.5:         BOOL;
input8          AT %IX12.5:         BOOL;
input9          AT %IX14.7:         BOOL;
```

Figure 24. Declaration of variables linked to GPIO.

When the program was running, the output variables were turning on and off the LEDs, whereas the values of the input variables changed when the corresponding buttons were pressed. However, some of the buttons did not affect the variables they were linked to, even though they were wired and configured in the same way as the correctly functioning buttons. The reason for that was that not all of the GPIO input pins do have internal pull-up resistors enabled by default.

Creating a pull-up circuit was necessary in this case because the pins were wired to the ground through the buttons. In this case, voltage is supplied to the input pin through a large (~50 kOhms) resistor, which makes the input value *HIGH* by default. When the button is pressed the circuit is closed to the ground making the value switch to *LOW*.

Raspberry Pi has built-in resistors that can be connected to the pins when needed. Figure 25 shows the default state of the input pins. The pins with *fsel* value “0” are configured as inputs, and their *level* value shows whether they are high or low by default.

```
BANK0 (GPIO 0 to 27):
GPIO 0: level=1 fsel=0 func=INPUT
GPIO 1: level=1 fsel=0 func=INPUT
GPIO 2: level=0 fsel=4 alt=0 func=SDA1
GPIO 3: level=1 fsel=4 alt=0 func=SCL1
GPIO 4: level=1 fsel=0 func=INPUT
GPIO 5: level=1 fsel=0 func=INPUT
GPIO 6: level=1 fsel=0 func=INPUT
GPIO 7: level=1 fsel=0 func=INPUT
GPIO 8: level=1 fsel=0 func=INPUT
GPIO 9: level=0 fsel=0 func=INPUT
GPIO 10: level=0 fsel=0 func=INPUT
GPIO 11: level=0 fsel=0 func=INPUT
GPIO 12: level=0 fsel=0 func=INPUT
GPIO 13: level=0 fsel=0 func=INPUT
GPIO 14: level=0 fsel=0 func=INPUT
GPIO 15: level=1 fsel=0 func=INPUT
GPIO 16: level=0 fsel=0 func=INPUT
GPIO 17: level=0 fsel=0 func=INPUT
GPIO 18: level=0 fsel=0 func=INPUT
GPIO 19: level=0 fsel=0 func=INPUT
GPIO 20: level=0 fsel=1 func=OUTPUT
GPIO 21: level=0 fsel=1 func=OUTPUT
GPIO 22: level=0 fsel=0 func=INPUT
GPIO 23: level=0 fsel=0 func=INPUT
GPIO 24: level=0 fsel=0 func=INPUT
GPIO 25: level=0 fsel=0 func=INPUT
GPIO 26: level=0 fsel=1 func=OUTPUT
GPIO 27: level=0 fsel=0 func=INPUT
```

Figure 25. Default state of GPIO input pins.

It can be seen that the IO pins have different values by default. The pull-up resistors of Raspberry Pi cannot be controlled from CODESYS. Therefore, they needed to be enabled on the microcontroller itself. The following lines were added to /boot/config.txt file, as shown in Figure 26.

```
#GPIO
#Enable pull-up resistors
gpio=5,6,13,12,19,16,22,23,25=pu
```

Figure 26. Enabling pull-up resistors on Raspberry Pi.

The commands in this file are executed when the microcontroller boots up. The line above enables pull-up resistors for stated GPIO pins, which makes the reading of the button states consistent and reliable.

3.3.2 Additional IO modules

Raspberry Pi provides a limited number of GPIO pins, which might not be sufficient for connecting external devices. What is more, the maximum input voltage for the pins is 3.3 V. Therefore, it was beneficial to extend the IO by adding external modules. In this solution, I2C digital and analogue modules from Horter were used. They were connected to Raspberry Pi via Horter I2C repeater, which "raises the level of the SCL and SDA pins so that

they are recognized clean 5V I2C slaves to 5V" (Kit I2C repeater for Raspberry Pi, n.d.).

Digital and analogue input modules were connected in a daisy chain which could be extended with more modules if needed. The modules were added to the Inspector RPi project as I2C devices. For the controller to identify the modules, their I2C ids needed to be found. Figure 27 shows checking the IDs of the available I2C device on Raspberry Pi with the *i2cdetet* tool.

```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- 08 -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- 38 -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 27. i2cdetect tool.

The values in the Figure 27 were converted from the hexadecimal to decimal numeral system and added to the project.

Finally, the input variables for I2C input modules were declared with memory addresses provided in the module configuration as shown in Figure 28.

```
(*Digital I2C inputs*)
i2Cbyte         AT %IB10:           BYTE;
(*Analog I2C inputs*)
analogCh1       AT %IW0:            WORD;
analogCh2       AT %IW1:            WORD;
analogCh3       AT %IW2:            WORD;
analogCh4       AT %IW3:            WORD;
analogCh5       AT %IW4:            WORD;
```

Figure 28. Declaration of the variables linked to GPIO.

4 TESTS AND IMPROVEMENTS

When the Inspector code in the Raspberry could finally be compiled, it was important to test it upon connecting to the web client application. This chapter describes how the tests were conducted as well as their results. Furthermore, it portrays the issues that were detected during tests and the actions that were taken to prevent them.

4.1 Connection to Inspector Web and testing

Inspector RPi was tested upon the Office instance of Inspector Web deployed on InSolution's server which is normally used for development and debugging purposes.

Firstly, to establish the connection between the Raspberry Pi and the server, such parameters as the IP address, the host port and the client name were set to the CODESYS program. Moreover, the IP and MAC-addresses of the microcontroller were hardcoded in the program. With these parameters, Raspberry PI was able to connect to the server and initiate the communication over TCP.

Secondly, a virtual machine called "Raspberry Pi" was created in the Inspector Web Client application: the MAC-address and other parameters of the microcontroller were entered. Provided that the Web Client application knows the MAC-address, it expects a message from a device with this address and proceeds with communication with the "machine".

Lastly, the IO parameters were configured in the Web Client application. These configurations were necessary for associating the physical channels with the meanings of the input and output signals. Some of the channels set as inputs provided the Client application with information about the current state of the "machine" whereas other inputs were linked to the reason code buttons. The output channels were linked to the LEDs on the reason code box, which indicated the state of the "machine".

The setup described above was tested by sending the physical signals and checking their representation in the user interface of the Web Client.

The illustration of different machine states and reason codes sent from the Raspberry Pi can be seen in the screenshots below. Figure 29 shows the state view with a state chart and provides availability and utilization figures. State durations, comments and reason codes can be found in the table in Figure 30. The graph in Figure 31 shows the comparison of durations of different machine states.

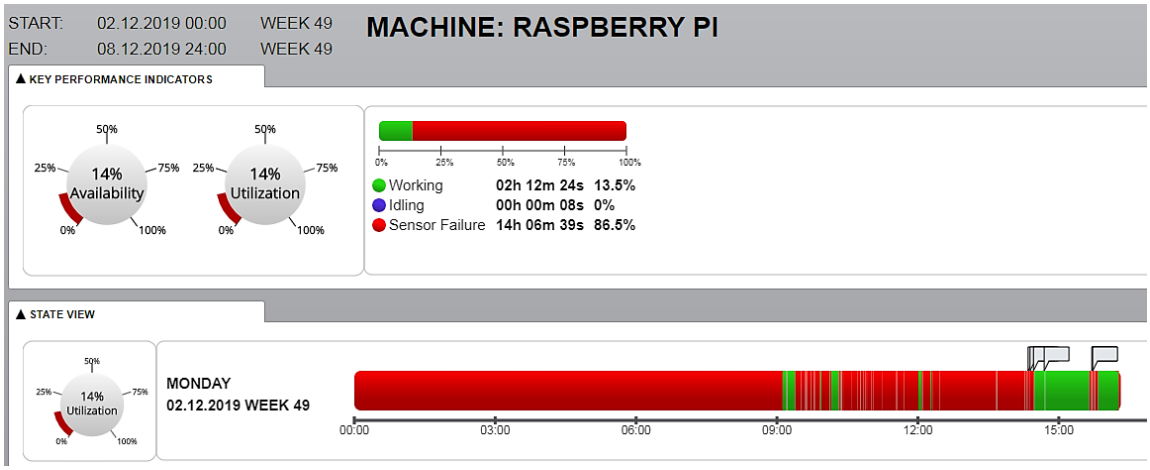

Figure 29. Inspector state view.

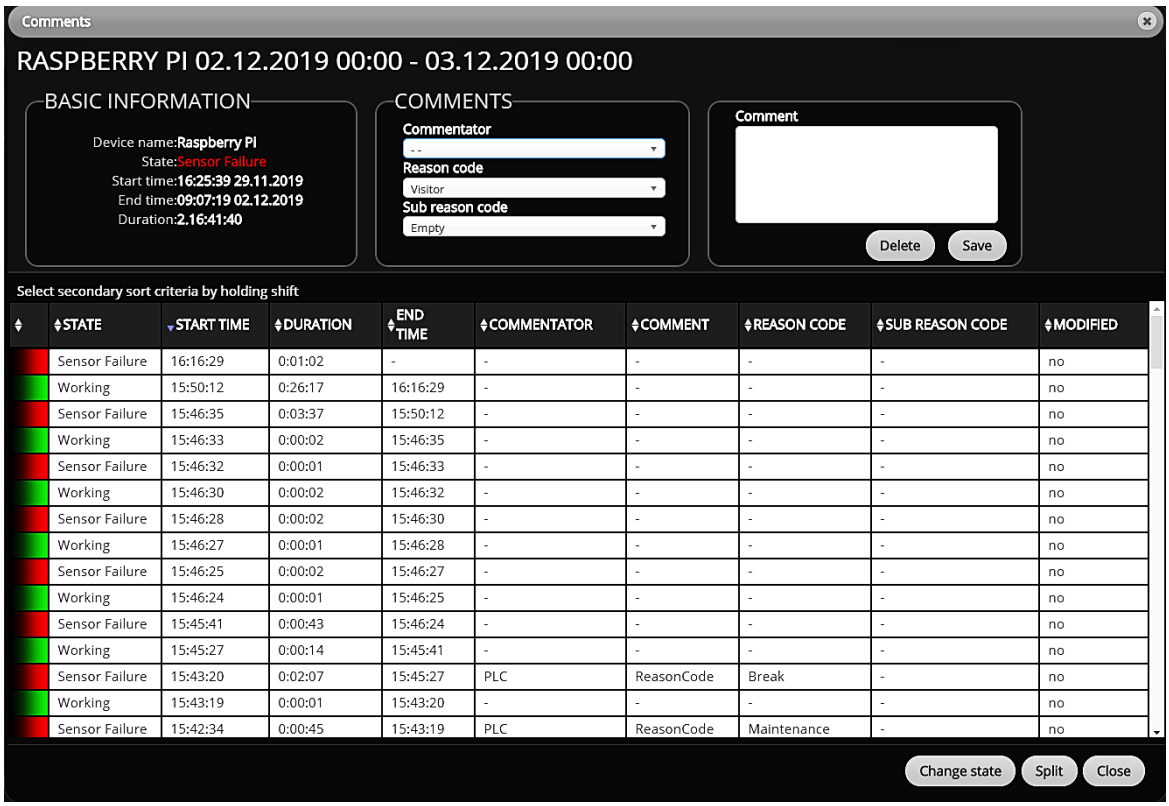

Figure 30. Inspector comments and durations.

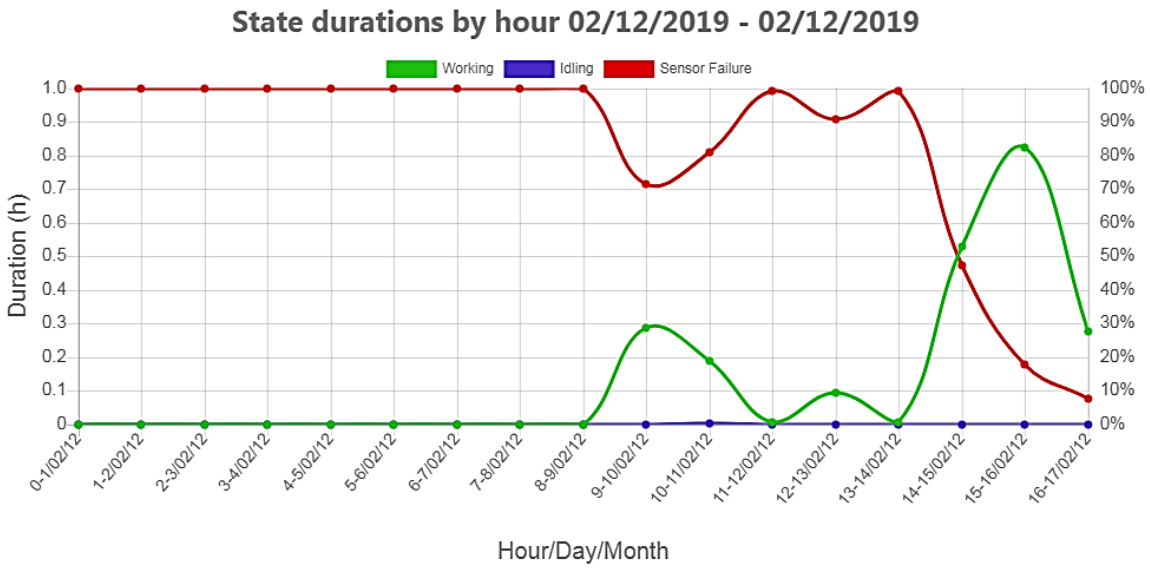

Figure 31. Inspector state charts.

What is more, analogue input was configured to test the CM data. The graph of the input value changing over time is shown in Figure 32.

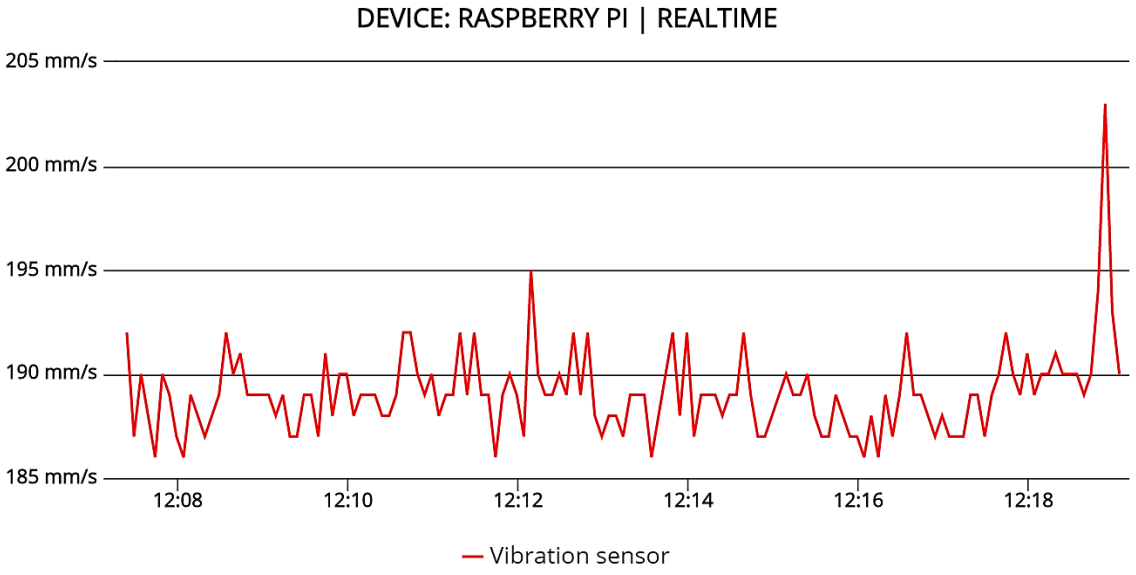

Figure 32. Inspector CM data.

Even though the numbers shown in the figures represent test input data and do not have any real meaning, they provide good examples of how production data can be collected and analysed using Inspector RPi. It was important to see that the application was reacting to the test inputs correctly and rapidly.

4.2 **Improving reliability of the data collection**

When collecting data on a real production site, there may be network or power failures. Moreover, sometimes the power at a plant is shut down when the production is stopped, for example, at night. It is also possible that the local network of the production site is slow or has connection interruptions. These situations are common for the production sites, where the Inspector tool is installed, and they can lead to data loss or corruption. The PLC Inspector code can take the potential network and power failures into consideration and, therefore, does not lose collected data in these situations. It was essential to provide the same reliability of the data collection for the Inspector RPi application.

4.2.1 Data collection in case of network failure

The logic in the program for Raspberry Pi of TCP/IP communication (see 3.2.3) states that in case the connection to the server is lost, send/receive operations are not performed while the controller is trying to re-establish the connection. However, when the connection issue was simulated by unplugging the Ethernet cable from the Raspberry Pi with running Inspector program, the result was different from expected.

After connecting the Raspberry Pi back, it was found that the CODESYS runtime service had been stopped on the microcontroller. The log of the service is shown in Figure 33.

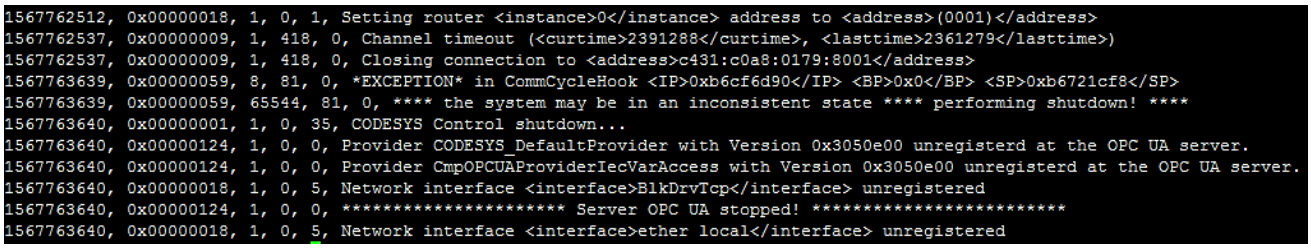

Figure 33. CODESYS log.

According to the screenshot above, after the communication timeout, the Raspberry Pi tried to close the connection, which caused an exception in CommCycleHook. That led to the system being in an inconsistent state and shutdown of CODESYS Control service was performed. Search on what CommCycleHook is and what could have caused the exception returned no results.

It was decided to stay logged into the Raspberry Pi program from the PC when the connection to the server is interrupted. By using breakpoints and watching online values of the variables it would possible to see what happens in the program before the exception.

The Raspberry Pi was connected to an Ethernet hub, which provided it with two wired connections: one to the office network, and one to the PC. That way, when the network cable was pulled out from the hub, the microcontroller lost its connection to the network and the server, but still could be accessed from the PC. However, with this setup, there was no exception when the connection to the server was lost and Raspberry Pi was able to reconnect to the server and send the data that was stored in the buffer. However, the exception repeated when the ethernet cable was disconnected from the Raspberry Pi. Consequently, this result meant that the exception only happened when there was not any network connection at all. It could be related to the fact that when there is no network connection, the microcontroller does not have an IP address, and attempting to connect to the server over a non-existing adapter drives the CODESYS Control service to an inconsistent state.

Based on that, it was decided to check the network adapters before trying to connect to the server. The library that was used for TCP communication, *SysSocket*, has function blocks for checking network adapters called *SysSockGetFirstAdapterInfo* and *SysSockGetNextAdapterInfo*. They return information on the MAC, IP and default gateway addresses of the device over its different adapters.

A small function block based on this functionality was created in a separate test program and was added to Inspector RPi after it was able to return correct network adapter information. The function block was tested both with and without network connection. To test how the program works in case there is no connection, a simple on-delay was used, so that the program was started, then the Ethernet cable was disconnected from the Raspberry Pi, and the adapter information check was executed one minute later. After connecting back to the microcontroller, it was possible to see the data returned by the function block during the offline state.

The adapter returned by the *SysSockGetFirstAdapterInfo* was always the Loopback adapter with IP address 127.0.0.1, independently on whether there was network connection or not. The *SysSockGetNextAdapterInfo* returned the correct information about the connection to the office network over the Ethernet cable when it existed. The third adapter belonged to the Wi-Fi connection. And when the connection to the network did not exist, zeros were returned for the IP and default gateway addresses. This way the network adapter information could be checked before trying to connect to the server without the possibility of causing an exception.

Moreover, retrieving the adapter information is benefitting for the Inspector application as it allows the Raspberry Pi to continue communication with the server even the IP address of the device is changed. It is possible because Inspector web-application identifies the connected machines by their MAC addresses, and also updates their IP

address to the database and GUI, which can be useful for network configuration and debugging purposes.

The function block was added to the Inspector code with the name “GetAdapterInfo” and has is called in two places. Firstly, in the main program during the initialisation of the program for obtaining the MAC address of the device. Secondly, in TCP_IP_CONNECTION function block before trying to connect to the server to check whether the second adapter exists.

At the end of the created function block several conversion operations were performed to obtain the MAC, IP and default gateway addresses in string format which helped to integrate the function block into the Inspector program. The full code of the function block can be found in Appendix 5.

After adding the *GetAdapterInfo* function block to the program, the test with unplugging the Ethernet cable from the Raspberry Pi was performed again. The microcontroller was able to reconnect to the server successfully without losing the data in the buffer, which means that the problem was solved.

4.2.2 Retaining variables in case of power failures

If the power is disconnected from the Raspberry Pi when there is data in the send and receive buffers, the data will be lost unless it is retained. The PLC Inspector has the buffer and other important variables declared with the keyword RETAIN.

Beckhoff PLCs write retain variables in the Non-Volatile Random Access Memory (NOVRAM, can also be referred as NVRAM) area, which is a specific memory component for persistent storage of data in a flash ROM (Read-Only Memory) but additionally has no write-cycle limitation. A capacitor, integrated into the NOVRAM chip, supplies the energy for copying recent data from the also internally integrated RAM to the ROM section during external power-loss situations. The application itself (in this case TwinCAT) writes only to the RAM section of the IC, cyclically. Storing variables in the NOVRAM memory allows their values to be retained in case of uncontrolled termination, application reset cold and program download. (Remanent Variables - PERSISTENT, RETAIN, n.d.), (Generic NOV-RAM, n.d.).

Raspberry Pi does not provide a NOVRAM memory area or UPS, which means that the data might be lost in case of a power failure, even if it is declared as retain or persistent. When performing Inspector tests, both retain and persistent data was saved by the microcontroller, only when it was shut down via the command line, and it had enough time to save the values. (Remanent Variables - RETAIN, PERSISTENT, n.d.).

Therefore, a Raspberry Pi would require additional hardware or an uninterruptible power supply (UPS) to save the values of the variables declared as PERSISTENT or RETAIN in case of power failure.

However, CODESYS provides two more mechanisms for data persistence besides Declaration of VAR Persistent: *Persistence Manager* of the Application Composer and *Recipes.* Both of these mechanisms do not require UPS or NOVRAM and therefore could be used to retain data on Raspberry Pi. It was decided to use the Persistence Manager, as its usage appeared simpler and more comprehensive then Recipes'. (Data Persistence, n.d.)

The Persistence manager requires the Application Composer which is a development tool for applications that use recurring function blocks. Application Composer was downloaded from CODESYS Store and a free licence for it was obtained.

In the *Modules* tab, a new Persistence Manager was created with a Persistence Channel linked to it, as shown in Figure 34.

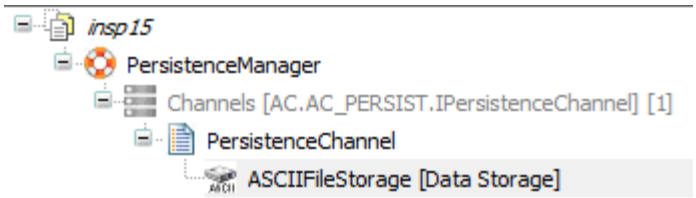

Figure 34. Persistence manager in Inspector RPi project

Variables that needed to have their values retained in case of power failure were added to the persistence channel by adding the following attribute to their declaration:

{attribute= 'ac_persist' :='PersistenceChannel'}

The channel was configured to save data every time changes are made to the variables and read their values during initialisation of the application.

When the new implementation of the data persistence was tested in the Inspector tool, the retain data was not lost even when the power cable was disconnected from the microcontroller. That proves that the Persistence Manager is an effective solution for making Inspector RPi more reliable.

4.3 **Prevention of cycle time increase in case of connection problems**

After the issue of CODESYS Runtime Service shutting down on losing network connection (see 4.2.1) was fixed, another problem was detected. When the connection to the server was lost and the Raspberry was making attempts to reconnect, the cycle time of the program increased dramatically: from around 0.5ms up to 6s. It was caused by the fact that the function block tries to establish the connection for a few seconds before it returns timeout error, and TCP communication was performed in the same task with the rest of the program. Thus, the TCP connection function block was slowing down the whole program cycle. What is more, it could be seen that the LEDs that normally were blinking fast to indicate that the Raspberry Pi is trying to connect to the server, were blinking much slower due to the delay in the task.

To eliminate this problem, it was decided to put TCP/IP communication to a separate task. A new task was created for communication where the *CONNECT, SEND* and *RECEIVE* function blocks were called continuously after the initialization of the program. Thus, even if the execution of the communication function blocks was slow, it did not affect the rest of the program, including managing of the LED outputs, and the program execution proceeded at normal speed.

5 **CONCLUSION**

The main goal of the thesis project was achieved successfully: Inspector PLC code was transferred to the CODESYS environment and executed on a Raspberry Pi microcontroller. With the adjustments, described in chapters 3 and 4, the new Raspberry Pi application was able to meet the requirements for the Inspector PLC application. Therefore, Raspberry Pi can become a replacement for a PLC in the Inspector tool.

The implementation of a PLC code on a Raspberry Pi in the CODESYS environment consisted of three main parts: firstly, software installations and preparations on the computer and Raspberry Pi, secondly, adjusting the program and its references to be compilable for CODESYS, and lastly, resolving the issues related to the hardware and improving the reliability of the application.

Due to a lack of documentation and research on the topic, finding suitable libraries and functionality was only possible by searching through CODESYS official documentation and experimentation with different approaches. Multiple tests and examinations were necessary to find the best solutions and to discover potential bugs and issues. The problems that were found during the tests required, in some cases, considerable adjustments to the program.

However, it was proved that a Raspberry Pi microcontroller with the CODESYS Control module can be used as a replacement for an industrial PLC. Whereas it is still not recommended to use Raspberry Pi for industrial solutions that require high precision or which control heavy machinery, the microcontroller can be a sufficient replacement for a PLC with smaller applications such as data collection, monitoring, testing, etc. as it proved to be for the Inspector tool.

Using a Raspberry Pi instead of an industrial-grade Beckhoff PLC will allow the case company, InSolution Oy, to significantly reduce the price of production of the Inspector tool without any losses in the reliability and the functionality of the end product. Unlike Beckhoff PLCs, Raspberry Pi does not have strict limitations to the code size, which allows the application functionality to be extended freely. The Inspector RPi application is more flexible in comparison to the Beckhoff PLC application since it does not depend on the controller model. Therefore, using Raspberry Pi as a controller would prevent issues that are faced when using Beckhoff PLCs.

As a result of the project success, InSolution Oy is planning to prepare Inspector Boxes with Raspberry Pi to be released for real-life customer applications. It is planned to make the first batch of 10 pieces and to conduct stress-tests to ensure the durability of the new product. The application can also be modified and improved over time by the author of the thesis or the other employees of the company, depending on future customer needs.

What is more, the CODESYS version of the Inspector PLC application, with necessary modifications, can be used on controllers from different manufacturers, other than Beckhoff, as CODESYS is a hardware-independent programming environment.

Finally, the knowledge gathered during the research and implementation of the thesis project can be useful for creating other automation projects with Raspberry Pi. As the thesis project has demonstrated, for some applications a Raspberry Pi can be a very cost-efficient alternative for an industrial-grade PLC, providing equivalent functionality and computing power.

REFERENCES

Arduino UNO WiFi Rev2. (2019). Retrieved December 21, 2019, from Arduino: https://store.arduino.cc/arduino-uno-wiFi-rev2

BC9020. (2017). Retrieved December 15, 2019, from Beckhoff: https://www.beckhoff.com/BC9020/

Bolton, W. (2015). *Programmable Logic Controllers* (6th ed.). Oxford: Elsevier Ltd. Retrieved from https://ebookcentral-proquest-com.ezproxy.hamk.fi/lib/hamk-ebooks/reader.action?docID=1985959&query=programmable%2Blogic%2Bcontrollers&ppg=1

Budimir, M. (2018, February 23). *What are IEC 61131-3 and PLCopen?* Retrieved from Motion Control Tips: https://www.motioncontroltips.com/iec-61131-3-plcopen/

Codesys. (2019). Retrieved August 25, 2019, from Codesys: https://www.codesys.com/

CODESYS Control for Raspberry Pi SL. (2019). Retrieved September 5, 2019, from CODESYS Store: https://store.codesys.com/codesys-control-for-raspberry-pi-sl.html?___store=en#Product%20Description

Communicatoin Protocol. (2019). Retrieved December 08, 2019, from Tecnopedia: https://www.techopedia.com/definition/25705/communication-protocol

Data Persistence. (n.d.). Retrieved October 27, 2019, from CODESYS Online Help: https://help.codesys.com/api-content/2/codesys/3.5.13.0/en/_cds_f_setting_data_persistence/

Generic NOV-RAM. (n.d.). Retrieved October 27, 2019, from Beckhoff Information System: https://infosys.beckhoff.com/english.php?content=../content/1033/tcsystemmanager/reference/FCNovRam.htm&id=

Hanssen, D. H. (2015). *Programmable Logic Controllers: A Practicap Approach to IEC 61231-3 using CODESYS :* (1st ed.). Chichester: Wiley.

History of the Programmable Logic Controller (PLC). (n.d.). Retrieved September 15, 2019, from PLCmentor: https://www.plcmentor.com/Articles/Newsletters/Programmable-Logic-Controller-PLC-History

I2C. (n.d.). Retrieved December 04, 2019, from Sparkfun: https://learn.sparkfun.com/tutorials/i2c/all

I2C – What's That? (n.d.). Retrieved December 04, 2019, from I2C-BUS: https://www.i2c-bus.org/

I2C Info – I2C Bus, Interface and Protocol. (2019). Retrieved December 03, 2019, from I2C Info: https://i2c.info/

Inspector Production monitoring. (n.d.). Retrieved September 5, 2019, from InSolution: https://insolution.fi/inspectoresite/

John, K.-H., & Tiegelkamp, M. (2010). *IEC61131-3: Programming Industrial Automation Systems* (2nd ed.). Berlin: Springer-Verlag.

Kabelová, A., & Dostálek, L. (2006). *Understanding TCP/IP : A Clear and Comprehensive Guide.* Brimingham, UK: Packt Publishing.

Katajisto, J. (2019).

Kit I2C repeater for Raspberry Pi. (n.d.). Retrieved December 1, 2019, from Horter & Kalb Online Shop: https://www.horter-shop.de/en/i2c-din-rail-modules/173-kit-i2c-repeater-for-raspberry-pi-4260404261155.html

LocalDateTime (FUN). (n.d.). Retrieved from CODESYS Online Help: https://help.codesys.com/webapp/D7UP-0urPzTyvbICO-3JvJB27cQ%2FLocalDateTime;product=Util;version=3.5.14.0

Micro800 Programmable Controller Family Selection Guide. (2019, March). Retrieved December 15, 2019, from Rockwell Automation: https://literature.rockwellautomation.com/idc/groups/literature/documents/sg/2080-sg001_-en-p.pdf

Micro820 Programmable Logic Controller Systems. (2019). Retrieved December 15, 2019, from Rockwell Automation: https://ab.rockwellautomation.com/Programmable-Controllers/Micro820#overview

Raspberry Pi. (2019). Retrieved September 5, 2019, from Open Source: https://opensource.com/resources/raspberry-pi

Raspberry Pi 3 Model B. (n.d.). Retrieved November 29, 2019, from RaspberryPi.org: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Remanent Variables - PERSISTENT, RETAIN. (n.d.). Retrieved October 25, 2019, from Beckhoff Information System:

https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/18014401038285451.html&id=

Remanent Variables - RETAIN, PERSISTENT. (n.d.). Retrieved October 27, 2019, from CODESYS Oline Help: https://help.codesys.com/api-content/2/codesys/3.5.12.0/en/_cds_vartypes_retain_persistent/

Rouse, M. (2019, July). *TCP/IP (Transmission Control Protocol/Internet Protocol)*. Retrieved September 14, 2019, from TechTarget: https://searchnetworking.techtarget.com/definition/TCP-IP

SSH (Secure Shell). (2018). Retrieved September 14, 2019, from SSH: https://www.ssh.com/ssh/?utm_source=s&utm_medium=nav&utm_campaign=head#sec-History-of-the-SSH-protocol

TwinCAT. (2019, January 15). Retrieved from Beckhoff: https://www.beckhoff.com/twincat/

What is a Raspberry Pi? (n.d.). Retrieved September 5, 2019, from Raspberry Pi: https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/

What is Arduino? (2019). Retrieved December 21, 2019, from Arduino: https://www.arduino.cc/en/guide/introduction

Why CODESYS? (2019). Retrieved August 31, 2019, from CODESYS: https://www.codesys.com/the-system/why-codesys.html

Ylonen, T. (1996). SSH - Secure Login Connections over the Internet. Proceedings of the 6th USENIX Security Symposium,. *USENIX*, (pp. 37-42). Retrieved from https://www.ssh.com/ssh/protocol/

Appendix 1

GET DATE AND TIME FUNCTION BLOCK

POU: GetDT

```
1   FUNCTION_BLOCK  GetDT
2   VAR_OUTPUT
3       res : DT ;
4   END_VAR
5   VAR
6       fbGetDateTime :  DTU . GetDateAndTime ;
7       fbTimezone :  DTU . SetTimeZoneInformation ;
8       periode : RTCLK . PERIODE ;
9       clk : BOOL := FALSE ;
10      clk2 : BOOL ;
11      tzInfo :  RTCLK . RTCLK_TIME_ZONE_INFO :=  DTU . GlobalConstants.
    gc_tziTimeZoneCET ;
12  END_VAR
13
```

```
1   IF NOT fbGetDateTime . xDone AND fbTimezone . xDone THEN
2       (*trigger for fbGetDateTime*)
3       clk := TRUE ;
4   ELSE
5       (* return result, add one hour during daylight  period*)
6       IF  periode = RTCLK . PERIODE . DAYLIGHT  THEN
7           res := fbGetDateTime . dtDateAndTime + T#1H ;
8       ELSE
9           res := fbGetDateTime . dtDateAndTime ;
10      END_IF
11      clk := FALSE ;
12  END_IF
13
14  (*get current time vaue  time*)
15  fbGetDateTime (
16      xExecute := clk ,
17      xDone => ,
18      xBusy => ,
19      xError => ,
20      eError => ,
21      dtDateAndTime =>,
22      ePeriode => periode );
23
24  (*set timezone info*)
25  fbTimezone ( xExecute := clk2 ,  tziInfo := tzInfo ) ;
26  IF NOT fbTimezone . xDone THEN
27      tzInfo . iBias  :=  TimezoneBias * 60 ;
28      clk2 := TRUE ;
29  END_IF
```

Appendix 2/1

TCP/IP CONNECT FUNCTION BLOCK

POU: TCP_IP_SOCKET_CONNECTION

```
FUNCTION_BLOCK  TCP_IP_SOCKET_CONNECTION
VAR_IN_OUT
    Flags : TCP_IP_Flags ;
END_VAR
VAR
    //SOCKET_CONNECT: FB_SocketConnect;
    //SOCKET_CLOSE: FB_SocketClose;
    GetAdapterInfo : GetAdapterInfo ;
    Step : INT := - 1;
    NextStep : INT ;
    PrevStep : INT ;
    StepChg : BOOL ;
    ErrorID :  STRING ( 32 ) ;
    oldErrorID :  STRING ( 32 ) ;
    closeCounter : WORD ;
END_VAR
```

```
CASE Step OF
0 : (* Waiting for connect command *)
    IF StepChg THEN
        Flags . Connected :=  FALSE ;
        IF PrevStep = 11 THEN
            Flags . ConnectStatus :=  'Waiting for connect command, timeout' ;
        ELSE
            Flags . ConnectStatus :=  'Waiting for connect command' ;
        END_IF
    END_IF
    (*Initialize connection*)
    //check if adapter exists
    GetAdapterInfo ( ) ;
    //proceed with connection only if adapter  exists
    //if trying to connect when Pi is not connected to the  network,
    //it will throw an exception an shut down codesys  runtime
    IF GetAdapterInfo . connected  THEN
        flags . iSocketHandle := SysSockCreate (
            iAddressFamily := SOCKET_AF_INET ,
            diType := SOCKET_STREAM ,
            diProtocol := SOCKET_IPPROTO_IP ,
            pResult := ADR ( flags . SocketHandleError  ) ) ;
        flags . rIP  REF=  flags . IPAddress ;
        SysSockInetAddr (
            szIPAddress :=  flags . rIP ,
            pInAddr := ADR ( flags . inaddr ) ) ;
        flags . stSockAddress . sin_addr := flags . inaddr ;
        flags . stSockAddress . sin_family := SOCKET_AF_INET ;
        flags . stSockAddress . sin_port :=
            SysSockHtons ( usHost := flags . Port ) ;
        NextStep := 10 ;
    END_IF

10 :     (* Connect to TCP/IP socket  *)
    IF StepChg THEN
        Flags . Connecting :=  TRUE ;
        Flags . ConnectStatus :=  'Connecting' ;
    END_IF
```

Appendix 2/2

TCP/IP CONNECT FUNCTION BLOCK

```
34          (*connect*)
35          flags . ConnectionStatus := SysSockConnect (
36                  hSocket := flags . iSocketHandle ,
37                  pSockAddr := ADR ( flags . stSockAddress ) ,
38                  diSockAddrSize := SIZEOF ( flags . stSockAddress ) ) ;
44
45
46          IF   flags . ConnectionStatus = CmpErrors . Errors . ERR_PENDING
47          OR  flags . ConnectionStatus = CmpErrors . Errors . ERR_SOCK_TIMEDOUT
48          OR  flags . ConnectionStatus = CmpErrors . Errors . ERR_TIMEOUT  THEN
49              NextStep := 0 ;
50          (*Error state, log error**)
51          ELSIF flags . ConnectionStatus <> 0 THEN
52              ErrorID :=  UDINT_TO_STRING ( flags . ConnectionStatus ) ;
53              NextStep := 999 ;
54          (* No Error *)
55          ELSE
56              NextStep := 11 ;
57              ErrorID := '' ;
58          END_IF
59          IF Step <> NextStep  THEN
60              Flags . Connecting :=  FALSE ;
61          END_IF
62
63      11 :      (* Connected*)
64          IF StepChg THEN
65              Flags . Connected :=  TRUE ;
66              Flags . ConnectStatus :=  CONCAT ( CONCAT ( CONCAT (
67                  'Connected to: ' ,  Flags . IPAddress ) , ':' ) ,
68                   WORD_TO_STRING ( Flags . Port ) ) ;
69          END_IF
70          IF NOT flags . Connected THEN
71              nextStep := 0 ;
72          END_IF
73
74      999 :     (* Error *)
75          Flags . ErrorCounter :=  Flags . ErrorCounter  + 1 ;
76          Flags . LastError :=  CONCAT ( CONCAT ( CONCAT (
77              'Connect error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,
78               Flags . ConnectStatus ) ;
79          Flags . ConnectStatus :=  CONCAT ( CONCAT ( CONCAT (
80                  'Error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,
81                   Flags . ConnectStatus ) ;
82          NextStep := 0 ;
83      END_CASE
84
85      (*Changing steps*)
86      StepChg := Step <> NextStep ;
87      IF StepChg THEN
88          PrevStep := Step ;
89          Step := NextStep ;
90      END_IF
91      IF Flags . Reset THEN
92          Step := 0 ;
93      END_IF
94
```

Appendix 3/1

TCP/IP SEND FUNCTION BLOCK

POU: TCP_IP_SOCKET_SEND

```
1   FUNCTION_BLOCK  TCP_IP_SOCKET_SEND
2   VAR_IN_OUT
3       Flags : TCP_IP_Flags ;
4       httpcom : HttpComm ;
5   END_VAR
6   VAR  CONSTANT
7       MaxSendBufferBytes : WORD := 20480 ;
8   END_VAR
9   VAR
10      Step : INT := - 1;
11      NextStep : INT ;
12      PrevStep : INT ;
13      StepChg : BOOL ;
14      ErrorID : STRING ;
15
16      SendBuffer :  ARRAY [ 1 .. MaxSendBufferBytes ]  OF  BYTE ;
17      BufferLen : WORD ;
18  END_VAR
19
```

```
1   CASE Step OF
2
3   0 : (* Idle *)
4       IF StepChg THEN
5           Flags . Sending :=  FALSE ;
6           Flags . SendStatus :=  'Waiting for send command' ;
7           ErrorID := '' ;
8       END_IF
9       IF Flags . Send   AND
10      NOT Flags . SendReceiveInProgress  AND
11       flags . Connected THEN
12          NextStep := 11 ;
13          Flags . SendReceiveInProgress := TRUE ;
14      END_IF
15
16  11 :     (* Send data *)
17      IF StepChg THEN
18          Flags . Sending :=  TRUE ;
19          Flags . SendStatus :=  'Sending data' ;
20      END_IF
21      BufferLen :=  HTTP_DataToByteBuffer ( SIZEOF ( SendBuffer ) ,
22                      ADR ( SendBuffer ) , httpcom . DataToSend ) ;
23      (*Send command*)
24      flags . BytesSent := SysSockSend (
25          hSocket := flags . iSocketHandle  ,
26          pbyBuffer := ADR ( SendBuffer )  ,
27          diBufferSize := BufferLen ,
28          diFlags := SOCKET_MSG_NONE ,
29          pResult := ADR ( flags . DataSendError ) ) ;
30      (*error or timeout*)
31      IF flags . DataSendError <> 0 THEN
32          ErrorID :=  UDINT_TO_STRING ( flags . DataSendError ) ;
33          NextStep := 999 ;
34      (*message sent*)
35      ELSIF flags . BytesSent <> 0 THEN
```

Appendix 3/2

TCP/IP SEND FUNCTION BLOCK

```
16              Flags . DataSendCounter :=  Flags . DataSendCounter  +  1 ;
17              flags . SendReceiveInProgress := FALSE ;
18              Flags . Send :=  FALSE ;
19              Flags . Sending :=  FALSE ;
20              NextStep  := 0 ;
21          END_IF
42
43      999 :     (* Error *)
44          flags . Sending := FALSE ;
45          flags . SendReceiveInProgress := FALSE ;
46          Flags . ErrorCounter :=  Flags . ErrorCounter  +  1 ;
47          Flags . LastError :=  CONCAT ( CONCAT ( CONCAT (
48              'Send error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,  Flags .
    SendStatus ) ;
49          Flags . SendStatus :=  CONCAT ( CONCAT ( CONCAT (
50              'Error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,  Flags .
    SendStatus ) ;
51          flags . Connected := FALSE ;
52          NextStep := 0 ;
53      END_CASE
54
55       (*Step change*)
56      StepChg  := Step <> NextStep ;
57      IF StepChg  THEN
58          PrevStep  := Step ;
59          Step  := NextStep ;
60      END_IF
61
```

Appendix 4/1

TCP/IP RECEIVE FUNCTION BLOCK

POU: TCP_IP_SOCKET_RECEIVE_FROM_HOST

```
FUNCTION_BLOCK  TCP_IP_SOCKET_RECEIVE_FROM_HOST
VAR_IN_OUT
    Flags : TCP_IP_Flags ;
    httpcom : HttpComm ;
END_VAR
VAR  CONSTANT
    TotalBufferSize : DWORD := 81920 ;
    MaxReceiveBufferBytes : WORD := 20480 ;
    StepMaxTime : TIME := TIME#2m0s0ms  ;
END_VAR
VAR
    Step : INT := 0 ;
    NextStep : INT ;
    StepChg : BOOL ;
    ErrorID : STRING ;
    TotalBuffer :  ARRAY [ 1 .. TotalBufferSize ]  OF  BYTE ;
    TotalBufferString :  ARRAY [ 1 .. 100 ]  OF  STRING ( 255 ) ;
    ReceiveBuffer :  ARRAY [ 1 .. MaxReceiveBufferBytes ]  OF  BYTE ;
    ReceivedBytes : DWORD ;
    recieve : BOOL := FALSE ;
    TotalReceivedBytes : DWORD := 0 ;
    ReceiveTimes : WORD ;
    iREcTime : INT ;
    irec : DWORD ;
END_VAR

```

```
CASE Step OF
0 : (* Idle *)
    IF StepChg THEN
        Flags . SendReceiveInProgress := FALSE ;
        Flags . ReceiveStatus :=  'Idle' ;
        ErrorID := '' ;
    END_IF
    IF Flags . Connected AND NOT flags .  SendReceiveInProgress
    AND flags . BytesSent <> 0 THEN
        (* New receive, reset *)
        MEMUtils . MemSet ( ADR ( httpcom . DataReceived )
                        , 0 , SIZEOF ( httpcom . DataReceived ) ) ;
        MEMUtils . MemSet ( ADR ( TotalBuffer ) ,
                            0 , TotalBufferSize ) ;
        MEMUtils . MemSet ( ADR ( TotalBufferString ) ,
                            0 ,  SIZEOF ( TotalBufferString ) ) ;
        ReceiveTimes := 0 ;
        TotalReceivedBytes := 0 ;
        NextStep := 11 ;
```

Appendix 4/2

TCP/IP RECEIVE FUNCTION BLOCK

```
3            ReceivedBytes := SysSockRecv (
4                hSocket := flags . iSocketHandle  ,
5                pbyBuffer := ADR ( ReceiveBuffer )  ,
6                diBufferSize  := MaxReceiveBufferBytes ,
7                diFlags  := SOCKET_MSG_DONTWAIT ,
8                pResult :=  ADR ( flags . DataRecieveError ) ) ;
9            (*Error State*)
10       IF  flags . DataRecieveError <> 0  AND  flags . DataRecieveError <> 518  THEN
11           ErrorID :=  UDINT_TO_STRING ( flags . DataRecieveError ) ;
12           NextStep  := 999 ;
13       (*Timeout, go to error state*)
14       ELSE
15           (*recieved a message*)
16           IF ReceivedBytes > 0  THEN
17               flags . SendReceiveInProgress := FALSE ;
18               flags . Receiving := false ;
19               IF TotalReceivedBytes < TotalBufferSize  THEN
20                   MEMUtils . MemCpy ( ADR ( TotalBuffer )  +
21                           TotalReceivedBytes ,
22                           ADR ( ReceiveBuffer ) , ReceivedBytes  +
23                           MIN  ( TotalBufferSize - TotalReceivedBytes -
24                               ReceivedBytes , 0 ) ) ;
25                   MEMUtils . MemCpy ( ADR ( TotalBufferString )  +
26                           TotalReceivedBytes ,
27                           ADR ( ReceiveBuffer ) , ReceivedBytes  +
28                           MIN  ( TotalBufferSize - TotalReceivedBytes -
29                                ReceivedBytes , 0 ) ) ;
30                   TotalReceivedBytes := MIN  ( TotalReceivedBytes +
31                            ReceivedBytes ,
32                            TotalBufferSize ) ;
33                   ParseByteBufferToHTTP_Data ( TotalReceivedBytes ,
34                       ADR ( TotalBuffer ) , httpcom . DataReceived ) ;
35                   Flags . DataReceiveCounter :=  Flags . DataReceiveCounter  +  1 ;
36                   ReceiveTimes := ReceiveTimes + 1 ;
37                   NextStep := 0 ;
38               ELSE (*buffer overflow error*)
39                   ErrorID := 'overflow' ;
40                   NextStep := 999 ;
41               END_IF
42           END_IF
43       END_IF
71
72   999 :    (* Error *)
73       IF StepChg  THEN
74           Flags . Receiving :=  FALSE ;
75           Flags  . Reset  := TRUE ;
76           Flags . ErrorCounter :=  Flags . ErrorCounter  +  1 ;
77           Flags . LastError :=  CONCAT ( CONCAT ( CONCAT (
78               'Receive error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,
79                Flags . ReceiveStatus ) ;
80           Flags . ReceiveStatus :=  CONCAT ( CONCAT ( CONCAT (
81               'Error with ID: ' ,  ErrorID ) ,  ', last status: ' ) ,
82                Flags . ReceiveStatus ) ;
83       END_IF
84           flags . Connected := FALSE ;
85           NextStep := 0 ;
```

Appendix 4/3

TCP/IP RECEIVE FUNCTION BLOCK

```
73    END_CASE
87
88    (*Step cange*)
89    StepChg := Step <> NextStep ;
90    IF StepChg THEN
91        Step := NextStep ;
92    END_IF
```

Appendix 5/1

GET NETWORK ADAPTERS FUNCTION BLOCK

POU: GetAdapterInfo

```
FUNCTION_BLOCK GetAdapterInfo
VAR_INPUT
END_VAR
VAR_OUTPUT
    connected : BOOL := FALSE ;
END_VAR
VAR
    FirstAdapterHandle : UDINT ;
    SecondAdapterHandle : udint ;
    size : UXINT ;
    mac : STRING ;
    i : BYTE ;
END_VAR
```

```
//getting first adapter, returnd loopback
size := SIZEOF ( AdapterInfo ) ;
FirstAdapterHandle := SysSockGetFirstAdapterInfo (
    pAdapterInfo := ADR ( AdapterInfo ) ,
    puxiAdapterInfoSize := ADR ( size ) ,
    pResult := ADR ( udiAdapterInfoError ) ) ;

//getting the second adapter, returns cable network connection
size := SIZEOF ( AdapterInfo ) ;
SecondAdapterHandle := SysSockGetNextAdapterInfo (
    hPrevAdapter := FirstAdapterHandle ,
    pAdapterInfo := ADR ( AdapterInfo ) ,
    puxiAdapterInfoSize := ADR ( size ) ,
    pResult := ADR ( udiAdapterInfoError ) ) ;

// if cable connection not found, check WiFi connection
IF AdapterInfo . IpAddr . ulAddr = 0 THEN
    SysSockGetNextAdapterInfo ( hPrevAdapter := SecondAdapterHandle ,
        pAdapterInfo := ADR ( AdapterInfo ) ,
        puxiAdapterInfoSize := ADR ( size ) ,
        pResult := ADR ( udiAdapterInfoError ) ) ;
END_IF

//convert IP address to string
sIPaddr := IoDrvEthernet . UDINT_TO_IPSTRING (
    udiIPAddress := MEM . ReverseBYTEsInDWORD (
        dwInput := AdapterInfo . IpAddr . ulAddr ) ) ;

//convert default gateway to string
sDefaultGateway := IoDrvEthernet . UDINT_TO_IPSTRING (
    udiIPAddress := MEM . ReverseBYTEsInDWORD (
        dwInput := AdapterInfo . DefaultGateway . ulAddr ) ) ;

//conver MAC address to string
mac := BYTE_TO_hexSTRING ( ADApterInfo . abyMac [ 0 ] ) ;
FOR i := 1 TO 5 DO
        mac := concat (
            concat ( mac , '-' ) ,
```

Appendix 5/2

GET NETWORK ADAPTERS FUNCTION BLOCK

```
34      FOR i := 1 TO 5 DO
35              mac := concat (
36                  concat ( mac , '-' ) ,
37                  BYTE_TO_hexSTRING ( AdapterInfo . abyMac [ i ] ) ) ;
38      END_FOR
41
42      DeviceID := mac ;
43      //true only if there is a valid network  connection
44      connected :=  AdapterInfo . IpAddr . ulAddr <> 0 ;
45
```