University College of Southeast Norway
Faculty of Technology
-
Master’s Thesis
System and Control Engineering
Spring 2016

Author: Sturla Rúnarsson

Open Source Hardware and Software Alternative to Industrial PLC

University College of Southeast Norway
Campus Porsgrunn/Faculty of Technology
Department of Electrical Engineering, IT and Cybernetics
Kjølnes ring 56
http://www.usn.no

This thesis is worth 30 ECTS

Abstract

The revolution of Internet of Things (IoT), Industrial Internet of Things (IIoT) and Industry 4.0 is approaching with new market opportunities for all kinds of smart devices. This thesis was about building such a device, an open source hardware and software alternative to industrial Programmable Logic Controller (PLC). The idea was to prove the concepts by building a prototype with a solid foundation that includes the best suiting communication protocol available (OPC-UA) and a modular functionality for ease of repair and customisation. The challenges were appropriate choices of open source hardware and software along with making sure that interactions between different parts of hardware were possible. That includes the communication protocols between modules and extensive programming needed in various programming languages. Several tests were then needed to validate the required communication speed and reliability requirements.

The prototype was developed with Open Platform Communications - Unified Architecture (OPC-UA) server module and four input/output (I/O) modules which include digital in, digital out, analog in and analog out. Raspberry Pi 2 was chosen as the System on Chip (SoC) hardware capable of running on Linux and hosting the OPC-UA server while Arduino Leonardo microcontrollers were chosen for the I/O modules. The OPC-UA server on the SoC hardware was programmed in Node.js on Linux while all I/O microcontrollers were programmed in a subset of C/C++. OPC-UA client in LabVIEW was developed for the majority of experiments while Matlab was used for data analysis.

The concept of building an open source PLC prototype was proven and its capabilities tested. The prototype proved to be stable in long time runs with no software/hardware crash on record. The communication speed from sensor to client (read) and client to actuator (write) was measured in LabVIEW with an average of under 10 ms. Only open source hardware and software was used, except for Raspberry Pi SoC OPC-UA server module which is not defined as an open source hardware by strictest definition. Modular I/O functionality was successfully implemented on a I^2C communication bus. The prototype shows potential for practical use and is ready for further development with emphasis on pin protections and upgrading I/O modules to industrialized standards for sensors and actuators.

Keywords: Prototyping, OPC-UA, Arduino, Raspberry Pi, I2C, PLC, Node.js, Open Source.

Contents

List of Figures

List of Tables

Preface

This thesis, "Open Source Hardware and Software Alternative to Industrial PLC," was carried out at University College of Southeast Norway (HSN) in collaboration with the engineering and consulting company EFLA. It was written under the supervision of Professor Nils-Olav Skeie. The FMH606 master's thesis is a mandatory 30 credit course in Systems and Control Engineering masters program in HSN. The signed task description can be seen in Appendix A.1. A separate "Project Abstract" document can be seen in Appendix A.2.

Many thanks to Professor Nils-Olav Skeie for the cooperation throughout this project and EFLA for consulting and providing resources needed.

Porsgrunn, 3. June 2016

Sturla Rúnarsson

Abbreviation

Abbreviation	**Definition**
AC	Alternating Current
ADC	Analog to Digital Converter
AE	Alarms and Events
ARM	Advanced RISC Machines
CA	Certificate authority
CLI	Command Line Interface
COM	Component Object Model
DA	Data Access
DAC	Digital to Analog Converter
DC	Direct Current
DCOM	Distributed Component Object Model
EU	European Union
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HDA	Historical Data Access
HMI	Human Machine Interface
I/O	Input / Output
I^2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IIoT	Industrial Internet Of Things
IoT	Internet Of Things
NIST	National Institute of Standards and Technology
OPC	Open Platform Communications
OPC-UA	Open Platform Communications - Unified Architecture
OSI	Open Systems Interconnection
PC	Personal Computer
PKI	Public Key Infrastructure
PLC	Programmable Logic Controller
PWM	Pulse Width Modulation
SCADA	Supervisory Control And Data Acquisition
SD	Secure Digital
SOA	Service Orientated Architecture
SoC	System on Chip
SPI	Serial Peripheral Interface
TCP/IP	Transmission Control Protocol / Internet Protocol
USB	Universal Serial Bus
Wi-Fi	Trademark for wireless standard IEEE 802.11x

Chapter 1

Introduction

Programmable Logic Controllers (PLC) have been used in the industry for sensory measurements and actuator control since the 1960's when it replaced the older technology of hardwired relay logic [1]. In simple terms the PLC measures sensory equipment in a process and then uses that information in a specific logic to control actuators in the process. New opportunities for PLC's and similar equipment are emerging with constant technology improvements and reduced costs in embedded devices, communication technologies, sensor networks and Internet protocols. Those opportunities offer improved information flow between devices, making them smarter in decision-making with added knowledge from other devices [2].

These new technology trends that mark the next transition to new types of PLC's are the Internet of Thing (IoT), Industrial Internet of Things (IIoT) and Industry 4.0 where devices communicate information between each other on larger scale through the Internet. These concepts are often talked about interchangeably as they are built on the same foundation but their definitions are different. IoT represents all kinds of consumer devices like wearable technologies, smart refrigerators and other home appliances that share data over the Internet. IIoT is defined the same way but for industrial applications. Industry 4.0 is a similar concept as IIoT that origins from the German government and identifies with the fourth industrial revolution, it is specifically focused on the manufacturing industry [3].

There are great advantages of IIoT over a traditional process network in the form of smarter decision making for individual devices. Those devices are smarter because they are able to operate with more information than before which helps them to make better decisions. These information can come from a database residing off-site, vendors, other factory, stock market or any other information available over the Internet that are beneficial to the device in place. Larger networks of processes can share information between them which can be analysed until patterns are found. Those patterns can then be used to predict, for example, wear and tear of components and devices in the process. The use of extra structured and unstructured data collection between processes has been called Big Data, which is enabled by IoT related technologies [4].

Most industry leaders and PLC vendors are preparing for IoT, IIoT, Industry 4.0 and Big Data. IBM, Intel, Cisco, GE, RTI and Rockwell Automation are among many vendors that are currently developing IIoT enabled technologies [5]. There are various markets that are affected by this technology including manufacturing, transportation, medical, home automation and many more.

The challenge for IoT related technologies are that it only supports first five layers in the Open Systems Interconnection (OSI) reference model which is a conceptual blueprint of how information is transmitted between any two points in a network [6]. That opens up questions about security and problems with vendor dependence which is defined at the application layer (layer 7) on the OSI model. Vendors need to use the same standards on layer 7 to be able to talk to each other. The communication protocol "Open Platform Communications Unified Architecture" (OPC-UA) has been identified as one of the enablers for IoT and IIoT technologies [7]. It resides completely on application level in the OSI model and is vendor independent [8, p. 350]. The security is built into the standard and offers data encryption, message signing and use of certificates for verification. It is only restricted to Transmission Control Protocol/Internet Protocol (TCP/IP) enabled devices which are already expected to be supported by IoT equipment [8, pp. 346-350]. OPC-UA is therefore considered a good communication protocol for the next generation of PLC devices. Note that Modbus TCP has been mentioned as a competitor to OPC-UA for the Industry 4.0 communication standard but this thesis will focus on OPC-UA [9].

The open source and prototyping community has made it possible for individuals and small teams to develop electronic devices without belonging to large corporations. That means that building prototypes has never been more accessible and is now possible with minimal efforts. Sparkfun Electronics and Adafruit Industries are examples of open source and prototyping communities that have large Forums where people can share information and learn from each other. Open source is not only related to software but also hardware. Open source hardware means that the hardware design is made public and anyone can make, modify, sell and distribute it. Open source software means that the source code is made publicly available and anyone can make, modify, sell and distribute it. Using an open source software and hardware can be beneficial to companies as it shortens development time, is cheaper and more secure. It shortens development time by allowing modifications to already built up software while shorter development time can lead to less overall cost. Security can be even more secure with open source because of the sheer manpower shared between the community when testing and fixing the code [10].

There are products available using parts of the desired functionality of IIoT enabled, open source, OPC-UA based and modular PLC. Both Siemens and Rockwell Automation are examples of vendors that sell embedded OPC- UA servers for their automation products which makes them IIoT compatible. Their products are however not open source. There are nevertheless open source projects with fully functional and certified PLC devices that were built upon open source microcontroller technology but lack an OPC-UA server, for example the Controllino and Industruino [11],[12]. Open source OPC-UA library built for Linux has been in development and tested on SoC platforms supporting Linux, but not in an open source PLC device [13]. The market therefore seems to lack a device that has all these functions in one package, that is IoT enabled, open source PLC with modular I/O and layer 7 support on the OSI model (OPC-UA). Modular I/O functionality is a part of most commercial PLC devices. It makes sure that the amount of inputs and outputs needed for each specific project is customisable and it will be easy to replace in case of failure. There has not, at the author best knowledge, been implemented an open source hardware/software PLC with inbuilt OPC-UA server and replaceable modular input/output (I/O) cards. The combination of IoT enabled PLC prototype with an open source OPC-UA server and modular approach where I/O cards can be stacked up as needed are the key concepts that add value to this project compared to other open source projects that are already out there.

This project is about using open source and prototyping resources to design and build a prototype of such a device and testing it for reliability and communication speed. The prototype is divided into modules, one power module that converts 230 V/Alternating Current (AC) to suitable Direct Current (DC), one OPC-UA server module that includes the Linux operation system and processing power and lastly four I/O modules. The four I/O modules include Digital In (DI), Digital Out (DO), Analog In (AI) and Analog Out (AO) with four ports each. The prototype will be built to be ready for early beta testing which means that it is ready for tests carried out by someone unconnected with the development process. The software for testing the prototype will be developed and a test plan document will be created that lists the testing procedures. The prototype will be tested thoroughly for communication speed and reliability. The communication speed is very important and is desired to be at least under 100 ms for reading sensors and writing to actuators. The reliability of the prototype hardware will be tested as well as the communication reliability and long term run reliability. The finished prototype should be a general purpose IoT PLC device that is easy to build upon and modify because of its open source nature.

In a report written in Accenture in 2015 it is estimated that OPC-UA in junction with IIoT could add $14.2 trillion to the worlds economy over the next 15 years [14, p. 5]. The motivation for this prototype is therefore not only academic but also highly relevant for the industry.

Chapter 2

System overview

The design strategy of the prototype was influenced by the current PLC design on the market, the open source community and by looking at the best industrialised communication protocols available (see Chapter 1). Simple overview of a LabVIEW OPC-UA client that is connected to the prototype with all key hardware components listed from 1-6 can be seen on Figure 2.1. Communication bus connects the I/O modules to the OPC-UA server module, a LabVIEW OPC-UA client then connects to the OPC-UA server over a network.

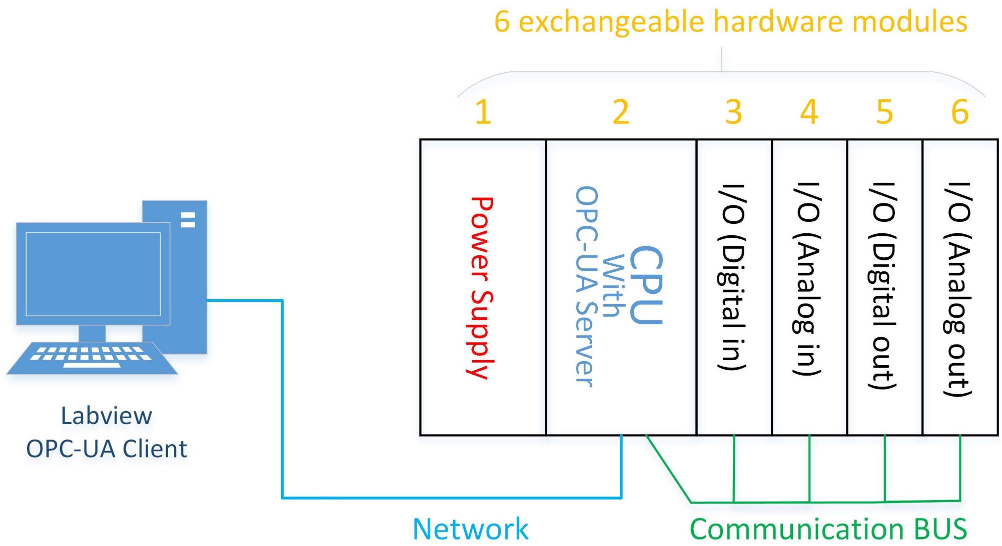


Figure 2.1: Overview of the PLC prototype setup with an OPC-UA LabVIEW client connected through a network. There are four different I/O cards that are connected with a communication bus to the OPC-UA server module. Each I/O has four ports.

All six hardware modules are within the boundary of the prototype. The hardware is carefully chosen to meet price, capabilities, interconnectivity and developed in a modular fashion. The exchangeable modular feature makes sure that the prototype can be customised for each project, depending on the amount and types of I/O cards needed. The modules are divided into standard modules and non-standard modules. Standard module means that it is not customised between projects and can be exchanged with no extra programming or change in software. The non-standard module has the main software that can vary between projects and therefore cannot simply be replaced without uploading the projects custom software again. The standard parts are the I/O modules and the power supply module, the only module that is customised for each project is the OPC-UA server module. Note that the OPC-UA server resides in the field devices, e.g. PLC's, IoT/IIoT devices and even sensors. Clients then read and write data from and to the servers, e.g. PC or a smartphone.

A Personal Computer (PC) running LabVIEW OPC-UA client on Windows 8.1 was used to test the prototype. The OPC-UA server is programmed on Linux on the OPC-UA server module. The I/O modules communicate to OPC-UA server module while controlling actuators and reading sensors. See Figure 2.2 for graphical presentation of connections between devices and key software.

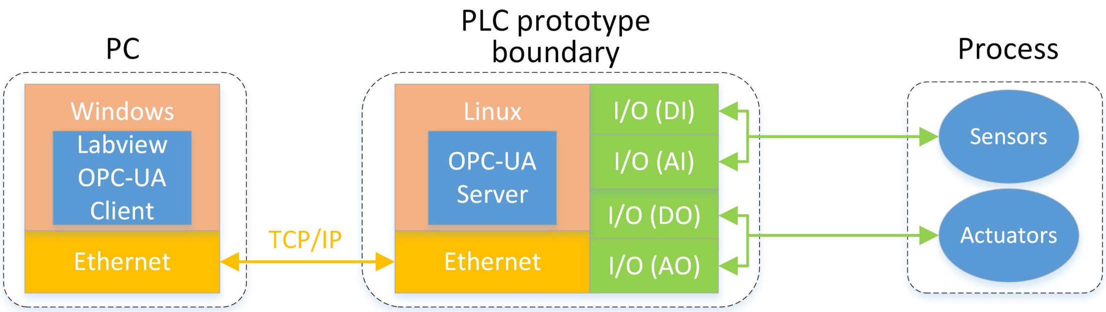


Figure 2.2: Graphical presentation of connections and key software between PC, prototype and process.

2.1 Requirements

There were necessary requirements defined that need to be fulfilled for the prototype to be practical in use. It should prove to be reliable in a longtime test-run, which is defined here as at least 10 days with no restarts or other failures. The prototype is supposed to be on beta stage after the completion of this thesis which means another person, not involved in the development, should be able to test it. That means that the prototype needs to have hardware shutdown and a start button. It also means that it has to be configured with services to automatically run the software on start-up. Indication lights for server running and successful power off will be included. There should be four I/O modules developed that operate on 0-5 V. Those I/O modules include digital in, digital out, analog in and analog out. Other I/O modules can be developed in future work but at this time it was considered important to develop these four standard ones. Software that is running on Secure Digital (SD) card can get corrupted if the hardware is not shut down correctly. A power off test will be applied to check whether the SD card, hosting the software on the prototype, will be corrupted.

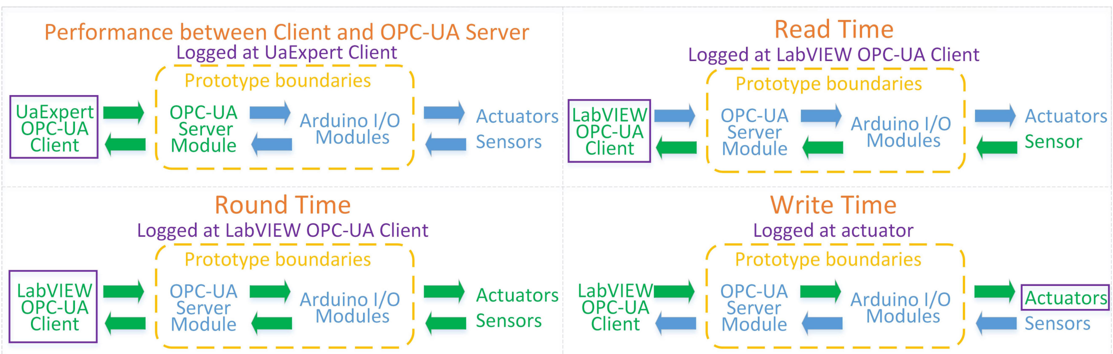


Figure 2.3: Overview of the measurement setup while logging the communication speed. There are four speed measurements; read, write, roundtrip and server to/from client.

It is recommended to observe Figure 2.3 for overview of the communication speed requirements and how the measurement setup is structured. LabVIEW OPC-UA client is used for measuring the read, write and roundtrip communication time while the UaExpert OPC-UA client is used for measuring the communication speed between OPC-UA server and OPC-UA client. UaExpert is a full-featured OPC UA client that is developed by Unified Automation. A free version with OPC UA Performance Plugin was used in this thesis [15]. The communication speed has to be fast enough to respond to quick changes in certain processes, that is defined here as at least under 100 ms. The exact communication speed needs to be measured precisely for both reading sensors and writing to actuators. The read time means the time from sensor and to OPC-UA client while the write time means the time from OPC-UA client to an actuator. The round time is also measured which means the round trip from OPC-UA client to actuator and back from sensor to OPC-UA client. The read and write time between OPC-UA server and OPC-UA client, exclusively, is measured as well. That way all the most important communication speed capabilities of the prototype is known. The difference between communication speed from various encryption settings and message signing has to be measured. That way it is known whether and how much encryption affects the communication speed.

The communication reliability is important because a wrong value can damage equipment that the prototype is controlling. It can also result in wrong values in the data logged. There should be an error detection on the communication bus between I/O modules and OPC-UA server module. That means that a communication protocol needs to be developed that fits the need of the prototype. The prototype should be tested for communication reliability between I/O modules and OPC-UA module by measuring the error frequency on the communication bus. The choice of hardware and software should be open source (see definition of open source in Section 1).

Collection of requirements:

- The read and write communication speed has to be under 100 ms, roundtrip communication speed should be under 200ms
- Communication speed between OPC-UA server and OPC-UA client should be known and be under 100ms
- Communication speed with various encryption and message signing should be measured
- The prototype should be reliable enough to function in long time runs, at least over 10 days in a single run
- Test the SD card for corruption when there is a power loss in the system
- Open source hardware and software
- Hardware shutdown and start button
- Prototype should be configured with automatic start-up services in case of a power loss and restarting the device
- Modular I/O design, interconnected on a communication bus
- Communication protocol with error detection between all I/O modules and OPC-UA server over the communication bus, measure the error frequency
- Four different I/O modules that are digital in, digital out, analog in and analog out
- Industrial grade communication protocol with IoT capabilities, OSI layer 7 support and inbuilt security (OPC-UA)

2.2 Modular design

Figure 2.4 shows a more detailed overview of the different modules types and connections. All I/O modules and the OPC-UA server module are connected together through the communication bus. The power module supplies power to all modules. There are four ports indicated on each I/O module that can be connected to various types of sensors and actuators within 0-5 V.

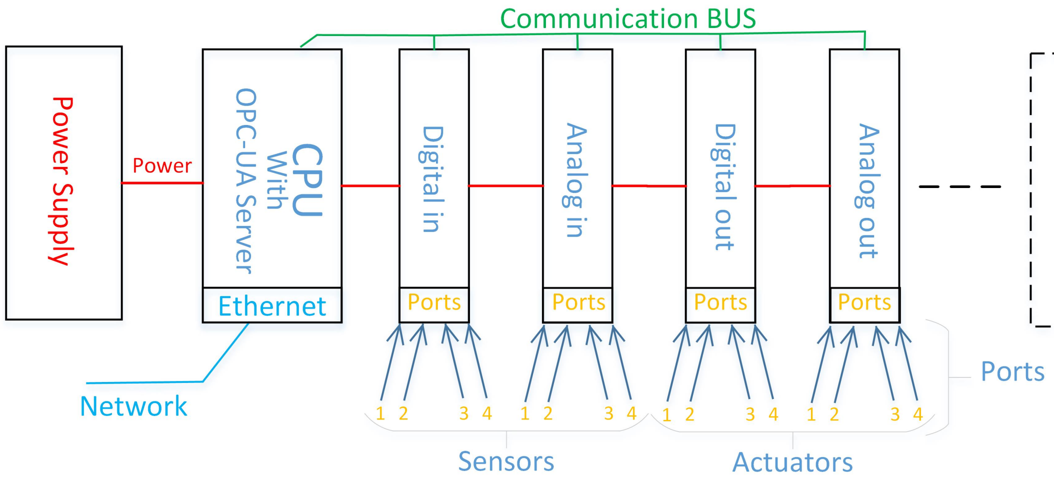


Figure 2.4: Detailed overview of the modular design with the four I/O modules types, showing four ports and connection with sensors and actuators. The OPC-UA server is connected with an Ethernet port to the desired network.

The four main types of I/O modules designed for the prototype include the following:

1. Digital in module with 4 inputs ports which receives 0-5 V
2. Digital out module with 4 outputs ports which supplies 0-5 V
3. Analog in module with 4 inputs ports which receives 0-5 V
4. Analog out module with 4 outputs ports which supplies 0-5 V

These four modules are defined as standard modules and should be able to be stacked in any combination whether one of them is only needed, many of the same type or all of them in various combinations. They are interchangeable and do not require extra customisation between different projects which makes them easy to replace in the case of a hardware failure. Note that while many sensors and actuators use 0-10 V and 4-20 mA it is the proof of concept that matters in this thesis and relatively easy to adapt the 0-5 V to 0-10 V or 4-20 mA later on.

The power module is responsible for supplying power to all components and modules in the prototype. It is defined as one of the standard modules and can be replaced without requiring extra customisation between different projects. Note that this power module was chosen for the amount of devices in the prototype, if other I/O modules are added it could mean other requirements for the power module.

The OPC-UA server module is defined as a non-standard module because it has all the custom software inside that can change between different projects. It needs to be configured for the amount of I/O modules in use and their types. All variables needed that are registered in the OPC-UA server need to be known and configured.

2.3 Communication protocol

The PLC prototype must be able to connect securely to various types of OPC-UA clients, e.g. Supervisory Control And Data Acquisition (SCADA) systems, Human Machine Interface (HMI) and IoT/IIoT devices. In industrial process and IoT/IIoT environments it is important that a secure, reliable and platform independent communication protocol is chosen. The OPC-UA protocol is open source, inbuilt security and support on OSI layer 7. It is considered one of the IIoT enabler for its security capabilities and multi-platform interoperability where different information technology systems and software applications can exchange data [7]. OPC-UA is therefore considered the best communication protocol for this system. Figure 2.5 shows example of clients connection to the PLC device via the OPC-UA protocol.

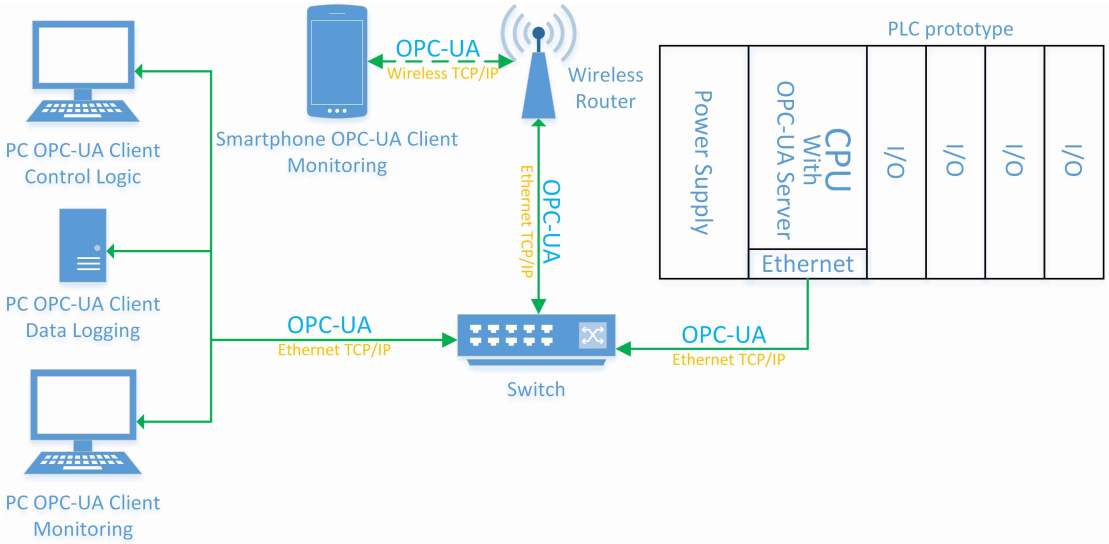


Figure 2.5: Example of clients connection to the PLC device via OPC-UA. It shows connections to different types of clients ranging from a smart-phone client to PC clients with both wireless and Ethernet connections.

2.4 Open source hardware and software

There are many advantages from designing the prototype with open source hardware and software (see open source definition in Chapter 1). First of all is the advantage of knowing the hardware and software thoroughly. The software is not restricted in any way and that means the ability to know the software on all layers. The open software makes sure that it is highly adjustable to modifications for different application purposes. When the hardware is open source it means that there is no vendor dependency and possible to modify the publicly available design to build custom hardware. The time saved by building software and hardware on pre-built libraries and software with open hardware design instead of building it from ground up can be enormous. With less time the cost is also considerably lower with less salary fees and fewer developers needed.

When both the hardware and software is open source it is possible to make an actual product from a prototype. It means no extra cost and royalties from 3rd party copyrighted solutions that could be needed for the project. After a prototype is realized it can be sent to production lines and sold under the developers brand. The open source and prototyping community is large, active and is of great help when developing innovating prototyping technology.

2.5 Use cases

To underline both the functionality and potential of this prototype two use cases are presented. Those use cases are made up and only presented to show how this device can be used instead of commercial PLC and possibly be easier in setup, more adaptable and much cheaper. This prototype can obviously be used in an industrial environment to read and control simple outputs and inputs, but it has much more potential for more specific projects. Those use cases will therefore not resemble a simple control or sensory reading in process environment but rather in use cases where conventional PLC units are less suitable.

2.5.1 Use case 1: Temperature and humidity monitoring in an office building

Let's assume that an office department wants to monitor the temperature and humidity at twelve places on one large office floor. They want to get all the values to a computer where the values are monitored and logged down for further analysis. In addition they want to have the option of showing the data on four information displays around the office. See Figure 2.6 for the office overview with PLC devices, display monitors and data logging clients.

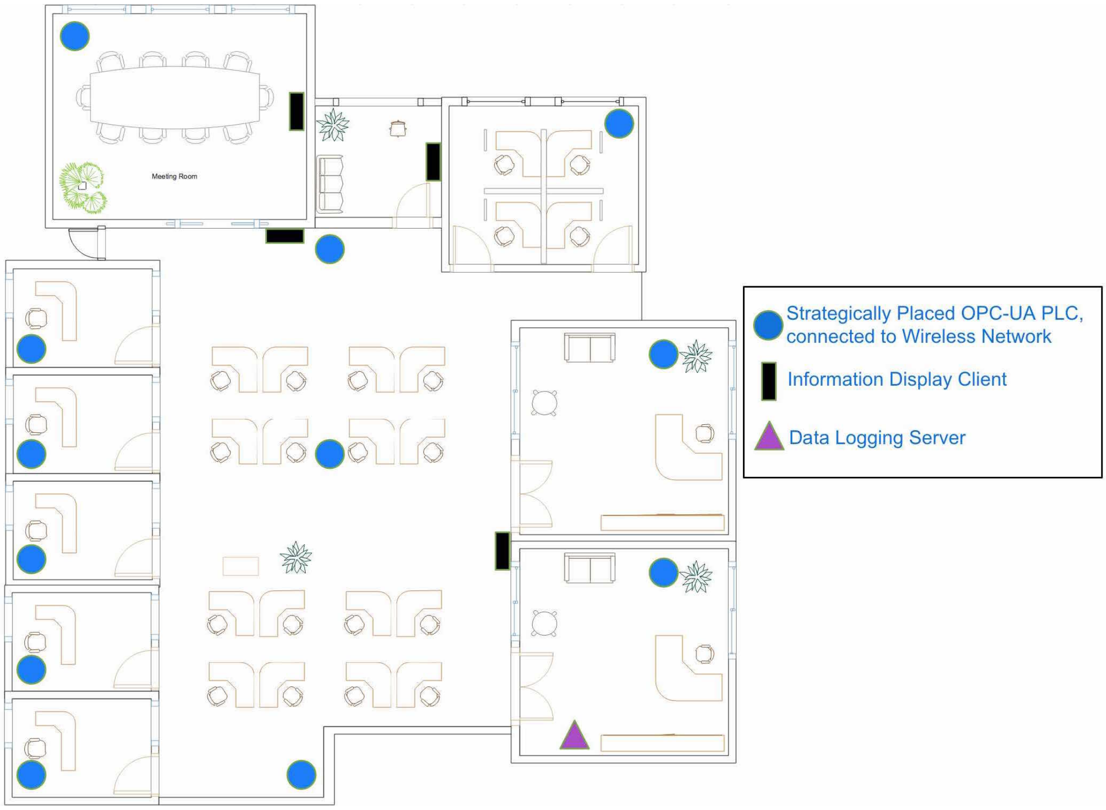


Figure 2.6: Overview of the office floor, showing the twelve PLC units, four information displays and one data logging server.

Solution:

Hardware: Set up the OPC-UA server module with one AI I/O and connect it to a commercial temperature and humidity sensor.

Software: Upload the default OPC-UA server program on to the server module and set it up for only one AI module. Set up the server to connect to the office Wi-Fi. Upload the default AI program to the AI module.

Installation: Distribute the devices around the office space and simply connect them to power.

Now all devices are communicating via Wi-Fi as OPC-UA servers and are easily accessible from any OPC-UA clients that are connected to the same Wi-Fi or over the Internet, independent of operating platform. The need for port configuration on the Router is needed for access outside the local network. That means that client connections for monitoring and server connections for logging and collecting data are simple, customisable and approachable. While estimating the cost, time and ease of installation it is logical to conclude that the prototype is better suited for this use case than using commercial PLC solutions.

2.5.2 Use case 2: Sensors monitoring and actuators control for the elderly to prolong their stay at own home

If the elderly are able to stay in their home longer before moving to a retirement home a mutual benefits arrives for both the government and the elderly. The benefit for the elderly is simply being able to take care of themselves longer and the government will be able to save some money. Let's assume that many factors will drive this project but only the monitoring and security part will be relevant for this use case. Let's also assume that Internet connection is available to all the residents involved in this project.

Moisture sensors can be used to make sure that there will not be a water damage because of running water not being turned off. The sensors will be placed where there is running water in the apartment. Light sensors will be used to get data from the resident whether he is able to remember turning off the light. Hinge sensor will be placed on the main door to see if the resident has isolated him self and not went out or gotten visitors. Heat sensor will be placed near the stove to see whether the resident forgot to turn the stove off. A power shutdown relay will then be connected to the monitoring system and therefore adds the ability to manually or automatically shut it down. Motion sensors can be placed around the apartment for security to see whether the resident has been able to move around the apartment. Panic buttons are then strategically placed around the apartment for the residents if they feel like they are in danger of some sort. See Figure 2.7 for overview of the apartment with sensors and actuators strategically placed at appropriate positions.

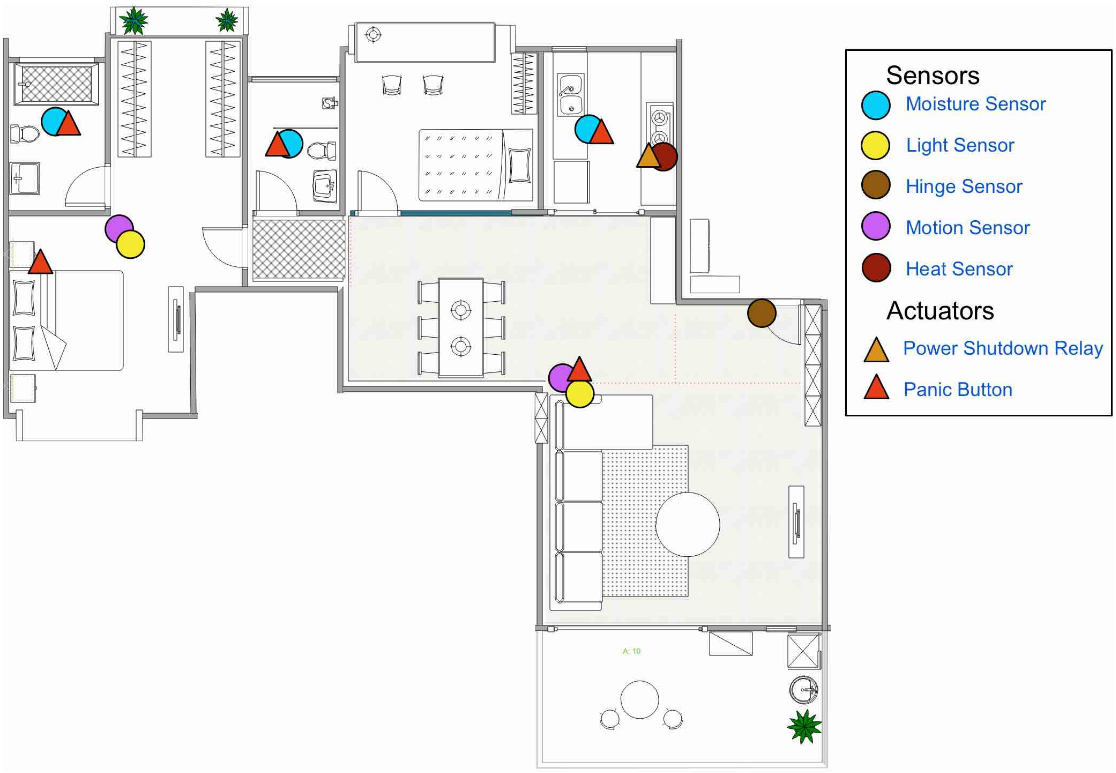


Figure 2.7: Overview of the senior citizen home floor plan, showing the position of all relevant sensors and actuators. It is a use case that let's senior citizens prolong their stay at their own home.

Solution:

Hardware: Set up the OPC-UA server modules with appropriate number of I/O modules for different types of sensor or actuators required.

Software: Upload the default OPC-UA server program on to the server module and set it up for the appropriate number of I/O modules. Set up the server to connect to the apartment Wi-Fi. Upload the default I/O programs to the modules.

Installation: Distribute the devices strategically around the apartment and connect to power. Set up the router with port-forwarding or similar solution for outer connection.

Now all devices are communicating via Wi-Fi as OPC-UA servers and are easily accessible from any OPC-UA clients that are connected to specific port on the apartment router, independent of operating platform. It allows, for example, a security company to monitor and control everything through the Internet. That means that all client connections for monitoring and server connection for data logging are simple, customisable and approachable. The same applies to this use-case as the previous one where cost, time and ease of installation conclude that the prototype is better suited than using commercial PLC solutions.

Chapter 3

Literature survey

3.1 Programmable Logic Controllers (PLC)

Programmable Logic Controllers have been called "the workhorse of industrial automation". They were introduced into the automation industry in the 1960's and went on to replace the older technology of the original hardwired relay logic. PLC had advantage over the older hardwired relay logic because of its small form factor that could replace hundreds of relays. It was also much easier for modifications or upgrades because PLC are programmable and no physical changes are needed for changed logic [1]. The simplest picture of the functionality of a PLC can be seen on Figure 3.1 where the PLC measures the sensory equipment in a process and then uses that information in a specific logic to control actuators in the process.

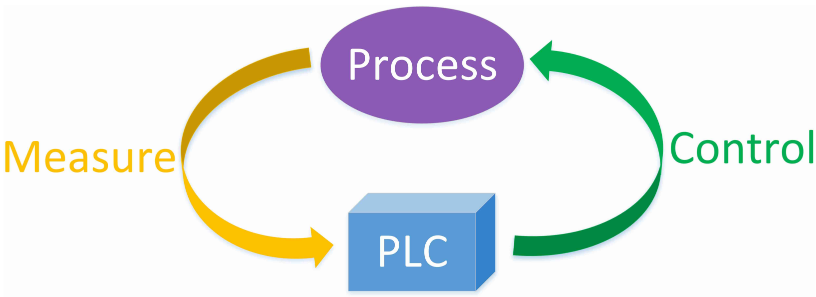


Figure 3.1: Basic description of a PLC functionality. It measures sensory devices from a process, processes the sensory information, then it controls devices in the process [16, p. 4].

In a market research study, done in 2015, it was estimated that with the slowing economic development and large number of PLC equipment reaching their life-cycle that the PLC market has largely become a replacement one. The customers are more frequently making their purchase decision based on the software capabilities and services instead of basing it on the hardware. The market has therefore evolved to push automation suppliers to evolve or lose market share [17]. That opens up some market opportunities where the capabilities of an open source hardware and software in junction with the prototyping community can offer a range of applications for production. The open source community gives individuals and small teams the ability to design and produce a commercial product instead of the power of prototyping only being in the hands of larger corporations.

The revolution of IoT/IIoT and Industry 4.0 is behind the corner where devices previously not connected to the Internet are being connected. That opens up opportunities for PLC with extended software and services capabilities both on the industrial and consumer market. The change from older processes to IoT/IIoT and Industry 4.0 technologies have great advantages in form of more information and smarter PLC devices. By having more information available the PLC devices will be able to take smarter decisions. Even though Industry 4.0 was initially a German government initiative it's ideology and strategy has been adopted in other countries as well. The European Union (EU) published a briefing in September 2015 that states its supports with Industry 4.0 through its industrial policy and does so with research and infrastructure funding [18]. The goal for Industry 4.0 is not to integrate the IT world to the process but rather to make machines more effective and easier to maintain[9].

IoT/IIoT and Industry 4.0 however only support layer 1-5 on the OSI model which means that the application layer 7 is missing. Some vendors build their own standards on the application layer which makes their devices not able to talk together between different vendors. Vendor independency is important that needs a communication protocol between those PLC devices that supports the OSI model layer 7. That is where OPC-UA comes in as a great vendor independent protocol that supports the application layer 7 in the OSI model. Vendors are constantly trying to adapt to the change in process technologies. Vendors that currently support OPC-UA servers in some of their PLC's are, for example, Siemens and Rockwell Automation.

3.2 Open Platform Communications (OPC)

OPC is about interoperability and standardisation for secure and reliable information exchange between all kinds of devices and services [19]. In this section the aspects of the old OPC standard versus the new OPC-UA standard are discussed as well as important parts of the OPC-UA protocol that is relevant to this thesis. The OPC Foundation manages a global organisation where vendors and its users collaborate to create data transfer standards that are secure, reliable, vendor independent and multi-platform. They create and maintain the specifications, offer certification testing to make sure of compliance with specifications and to collaborate with industry leaders [20].

OPC classic was released in 1996 and was the forerunner of OPC-UA [21]. OPC was pressured to evolve because of the dependency of Microsoft platform in OPC classic, security issues with Component Object Model (COM)/Distributed Component Object Model (DCOM) and problem with firewall configuration. OPC-UA was then released in 2008 and is now using cross-platform Service Orientated Architecture (SOA) instead of COM/DCOM [22]. See Figure 3.3 for a graphical representation of the cross-platform communication between OPC-UA clients and servers of different platform types. It has all the functionality of the older OPC classic specification and is backward compatible [23]. The Alarms and Events (AE), Data Access (DA) and Historical Data Access (HDA) servers in OPC classic have been simplified into one Unified Architecture that simplifies the integration significantly (see Figure 3.2).

The OPC Foundation announced, in April 2016, that the OPC-UA specifications have been made open source and publicly available. That was done to help eliminating roadblocks to the adoption of the OPC-UA technology [24]. The older DCOM specification, that allows COM components to work over a network, is considered firewall unfriendly because of it dynamically assigning TCP port to each executable process that uses DCOM objects [25]. It is not possible to configure dynamically changing ports to a firewall. If the firewall can not be configured and is turned off it makes the computer insecure and vulnerable to hackers and viruses. OPC-UA uses one or few static ports that can be configured in the firewall which makes the computer more secure.

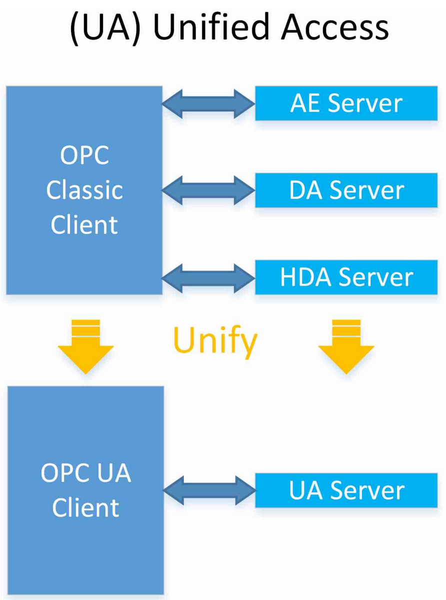


Figure 3.2: Unified Arcitecture simplification of DA, AE and HDA. It simplifies field integration and communication network [26, p. 11].

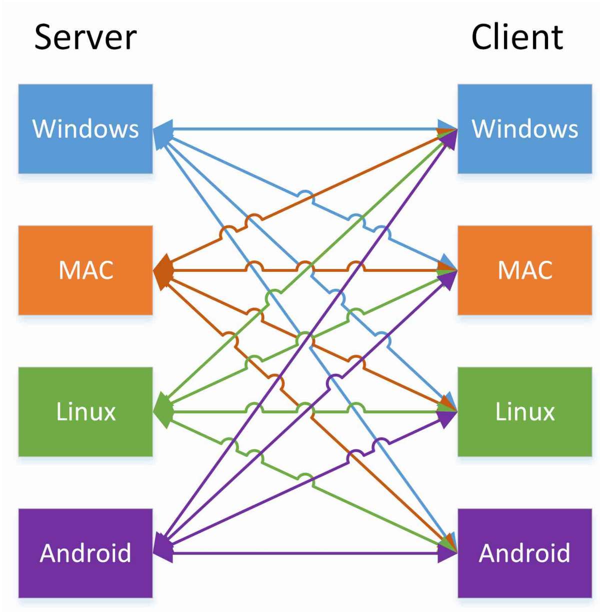


Figure 3.3: Shows OPC-UA platform interoperability where different operating systems and programming languages are able to communicate with each other [26, p:18].

In current times, with many OPC-UA solutions available, users are wondering when and if they should migrate their current OPC classic processes to OPC-UA. The new specifications overcome DCOM problems that makes OPC products more secure. It can therefore be beneficial in many processes to migrate to OPC-UA and is safe to assume that OPC UA will one day replace OPC classic [27].

3.2.1 OPC-UA

OPC-UA is a communication protocol that aims to achieve interoperability between all kinds of devices and services. The ability of embedding OPC-UA servers into microcontrollers offer range of possibilities and cuts out extra equipment for translating proprietary protocols to OPC classic. The support from vendors is extensive and has resulted in over 35.000 different OPC products from more than 4.200 suppliers [28]. When using OPC-UA in embedded hardware it can significantly reduce the complexity of the information exchange as seen on Figure 3.4.

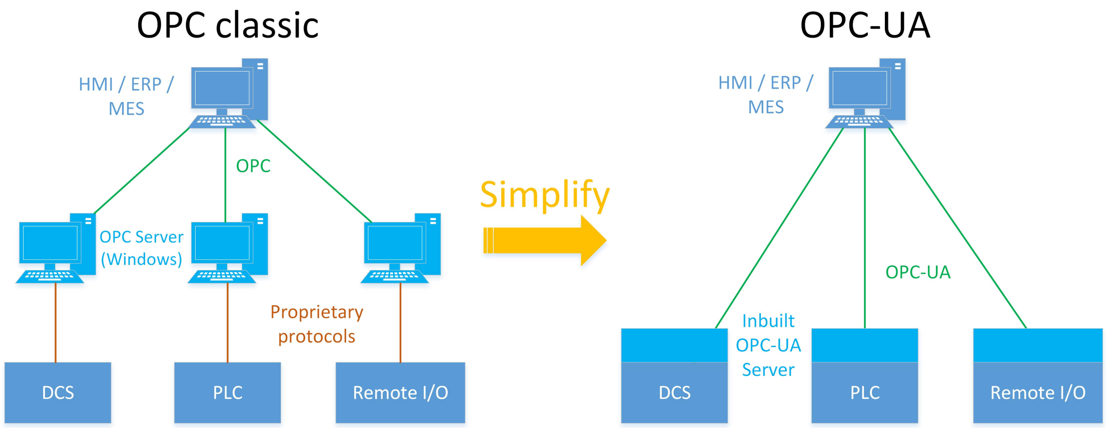


Figure 3.4: Overview of how the move from OPC classic to OPC-UA in embedded hardware can simplify the overhead. It shows that the complexity level can decrease which results in a more reliable system and simpler implementation. The older standalone servers are now inbuilt into the embedded devices in OPC-UA with no need for proprietary protocols [29].

OPC-UA is considered one of the enablers for IoT, IIoT and industrial 4.0[7], mainly for its ability to add support for the application layer 7 in the OSI model. It enables devices that are not traditionally connected to the Internet to connect safely and communicate within one, vendor independent and scale-able communication standard. These technologies are considered one of the most important drivers of the digital growth. Because of the service-orientated architecture and its inbuilt security it provides different devices of all platforms and sizes to exchange information securely and reliably [30].

3.2.2 Security

OPC-UA security involves authentication and authorisation, encryption of data and data-integrity via signatures. The most basic picture of the security of OPC-UA communication can be seen on Figure 3.5. It is recommended as side-note for the upcoming sections.

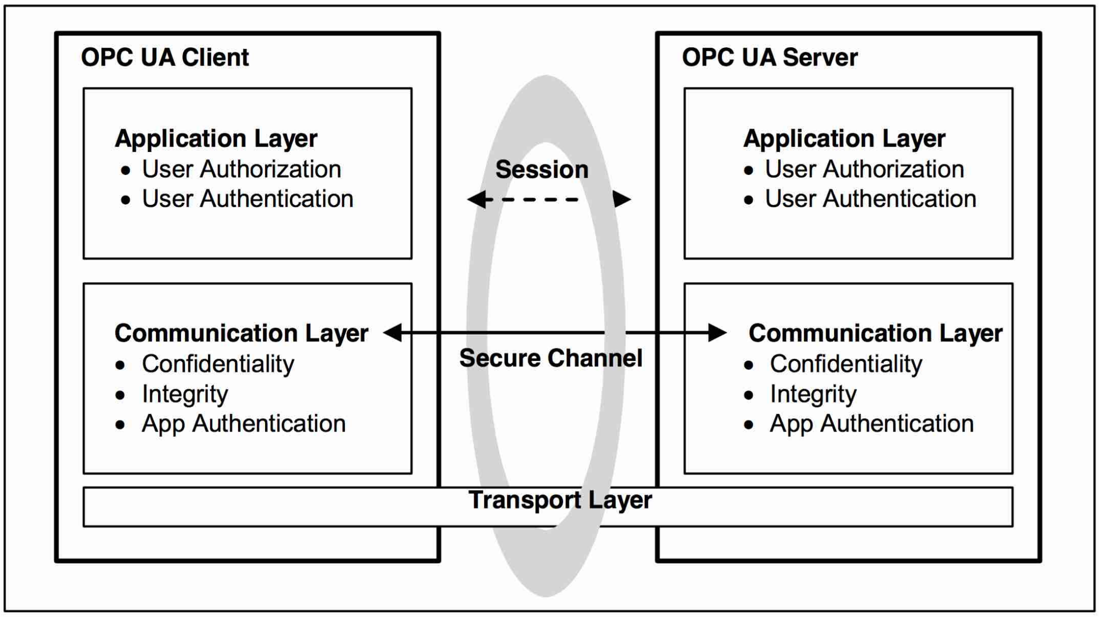


Figure 3.5: OPC UA security architecture in and between client and server, including user authentication and secure channel with encryption[31, p. 12] .

Authentication and authorisation

The session can be created on the application layer with three different authentication methods: Anonymous, user name/password and certifications. There are many certification standards available, X.509 is one of them and is specified as the certification method used in OPC-UA. Anonymous authentication is not recommended for security reasons. User name and password is configured in the server and the client is trusted if he has the right credentials. Using X.509 certification for authentication is recommended because it only allows clients that are specifically trusted by the server to connect. It tackles the security danger of a stolen user name and password. X.509 authentication technology consists of sets of private and public keys. The public keys are placed into certificate for distribution while the private key is protected. The authorisation of the read/write access level is also granted on the application layer depending on the clients purpose in the system. It is configured in the server for each individual user [32].

Confidentiality, integrity and application authentication

The confidentiality is configured on the communication layer and is done by encrypting the message. The security policies come in four configurations:

- None - Lowest security, no encryption
- Basic128Rsa15 - Medium security (128 bits), requires a Public Key Infrastructure (PKI)
- Basic256 - Medium to high security needs (256 bits), requires a PKI
- Basic256Sha256 - High security needs (256 bits), requires a PKI [33]

Note that security policies are expected to expire with time because of increasing computer processing capabilities. National Institute of Standards and Technology (NIST) recommends that keys with length under 2048 with security police Basic128Rsa15 and Basic256 should be upgraded in 2010 and that the policy should be deprecated in 2012 unless keys are over 2048 in length. Basic256Sha256 has no published end dates at the time of this thesis [33]. The integrity of the information is important for the receiver to receive the same message as the sender sent and that is done by having an unique identifier between messages.

The application authentication needs more detailed sequence diagram to show better overview of the security-architecture communication (see Figure 3.6). It shows how the client does endpoint request to the server and gets response from the server about its security and policies configurations. It also sends its server certificate to the client. The client then validates the server certificate by contacting Certificate Authority (CA). The client then asks the server for a secure channel and sends its certificate and private key which is encrypted with the server's public key. The server then validates the certificate with the CA. The secure channel is then opened between the client and server.

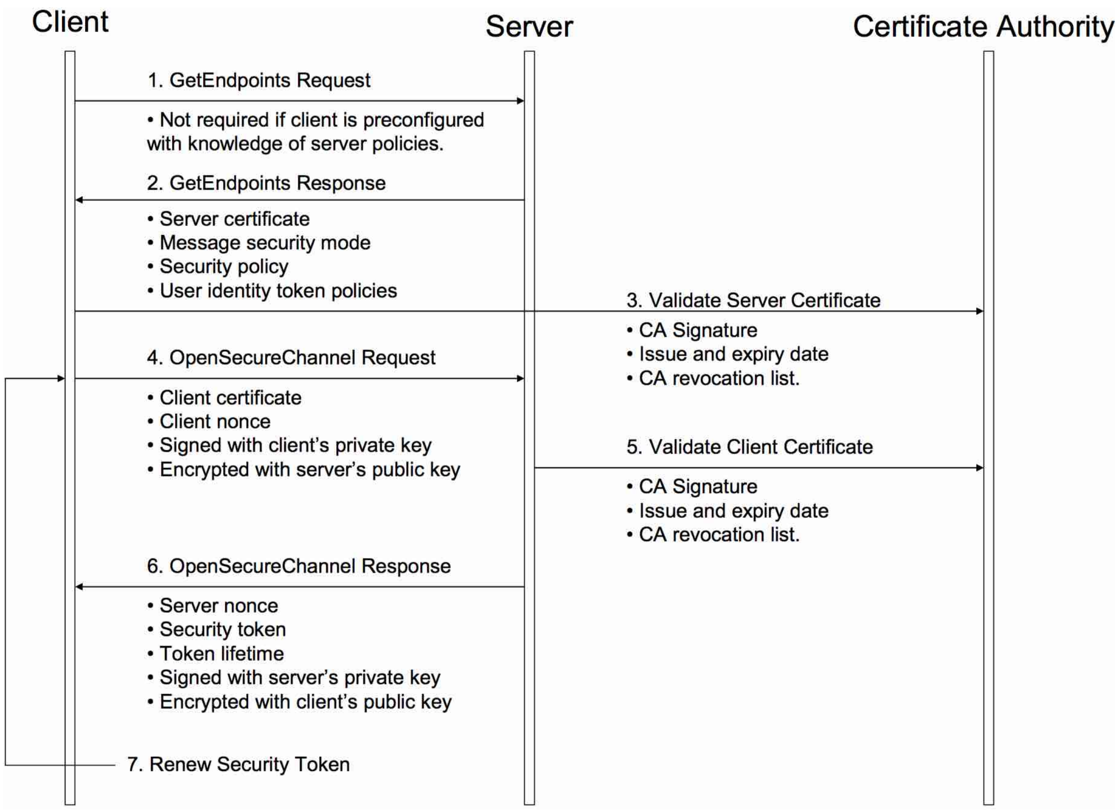


Figure 3.6: Sequence diagram of determining if an Application Instance Certificate is Trusted [32, p. 88].

Chapter 4

System description

In this chapter the choice of hardware components are justified and the selected hardware explained. The assembly of the prototype hardware is described and schematics are shown. The main software that is developed for the prototype is explained as well as the structure of the communication protocol between I/O modules and OPC-UA server module.

4.1 Hardware

Microcontrollers were considered the best choice for the I/O modules as they are cheap, reliable and designed for I/O applications. A System on Chip (SOC) hardware was chosen as the OPC-UA server module because of its capability of running Linux. Note that the choices of hardware do not necessary represents the best possible choices for a final product but are merely the best prototyping hardware to prove the concept. Only after the prototype shows a good potential for a product it is upgraded to hardware that represent better choices for mass production.

The process of choosing both the microcontroller and SoC for the prototype was done with these key features in mind:

1. Amount of support and size of online community
2. Open source hardware and software
3. Price
4. Extensible software and hardware add on
5. Communication Bus support

4.1.1 Choice of SoC for OPC-UA server module

There is a wide range of SoC hardware available and comparison of few of them can be observed in Table 4.1. The most important deciding factors were the amount of support, size of the online community as well as the open source nature of the hardware. The amount of support is crucial so that there is not the need to "reinvent the wheel" for every problem, that shortens the development time. The Raspberry Pi 2 had slight edge in comparison but the amount of support and price was the deciding factor. Note that Raspberry Pi is not an open source hardware by strictest definitions, even though it is often labeled as an open source. It is mainly because of the Advanced RISC Machines (ARM) architecture processor from Broadcom that is at the heart of the Raspberry Pi. It means that if someone wants to develop the board further it is necessary to contact, get permission and buy those ARM processors from Broadcom. Many other aspects of the Raspberry Pi are of open source nature and are publicly available. Note that the OPC-UA server is developed on Linux and can easily be ported to other hardware platforms that also support Linux. It means that it is easy to replace the SoC hardware if a new and better suited hardware, supporting Linux, will be available in the future. The closest competitor was the Beaglebone, it has fully open source hardware but it is expensive and lacking in support compared to Raspberry Pi [34]. The prototype does not need more powerful hardware than the Raspberry Pi 2 has to offer. The more power and extra features in the more expensive hardware platforms were therefore not necessary because the OPC-UA module is only supposed to run one, relatively light, OPC-UA server. Raspberry Pi is one of the most popular SoC hardware and has been sold in over 8 million units world-wide as of 29. February 2016, which is exactly 4 years after its release [35]. It has been directed towards educators, beginners and advance users, and has resulted in large support community containing tutorials and other helpful material. It is also notable that Raspberry Pi is currently used as a teaching tool in many courses in University Collage of Southeast Norway.

Table 4.1: Comparison between different types of SoC hardware with most important requirements listed. Note that pricing differs between sites and the hardware has different components, it therefore has arbitrary score. The community support was estimated by the amount of posts in the last 30 days and on the total posts available. Hardware and software support was determined by the available additions listed at the vendor's home sites.

SOC board	Open Source Hardware	Open Source Software	Community Support	Price	Hardware/Software Addition Support	Communication Bus support
Beaglebone	Yes	Yes	Medium	Very High	Medium	SPI / I2C
Banana Pi	Yes	Yes	Medium	Low	Medium	SPI / I2C
Arduino Yún	Yes	No	High	Very High	Very High	SPI / I2C
Raspberry Pi 2	Yes	Not by strictest standards	Very High	Low	Very High	SPI / I2C
Intel Edison	No	No	Medium	High	Low	SPI / I2C

4.1.2 Choice of microcontroller for I/O modules

The choice of microcontroller for the prototype was more straight-forward because the most popular ones for prototyping are almost all built upon the Arduino platform. Before Arduino the prototyping community was using other platforms like PIC processors but they were mostly considered for advanced users. Just like the Raspberry Pi the Arduino was made for educators and prototyping for beginners in mind. It has built a large community and support throughout the years since its launch in 2005 [36]. As of 2013, Arduino has registered over 700.000 official boards but it is estimated that there is one derivative or clone of Arduino platform per every official one [37]. Arduino is also used in many courses as a teaching platform for microcontrollers in University Collage of Southeast Norway. The most popular board is the Arduino UNO but for this project the Arduino Leonardo was chosen because it was the cheapest, full size Arduino board. Arduino Leonardo has completely open source hardware and software with publicly available schematic and reference design [38].

4.1.3 Choice of communication protocol between SoC and I/O modules

Arduino and Raspberry Pi 2 support both Serial Peripheral Interface (SPI) bus and Inter-Integrated Circuit (I^2C) bus. The communication bus that connects the I/O modules (Arduino) to the OPC-UA server module (Raspberry Pi) needs to handle multi-master connections. The reason for the multi-master requirement is that both the Arduino I/O modules and the Raspberry Pi have to be able to initiate communication. The SPI bus is only single master and therefore not suitable for this project. I^2C was therefore chosen as the best suited communication protocol. See Figure 4.1 for overview of the I^2C bus connections between the SoC hardware (OPC-UA server module) and the microcontrollers (I/O modules).

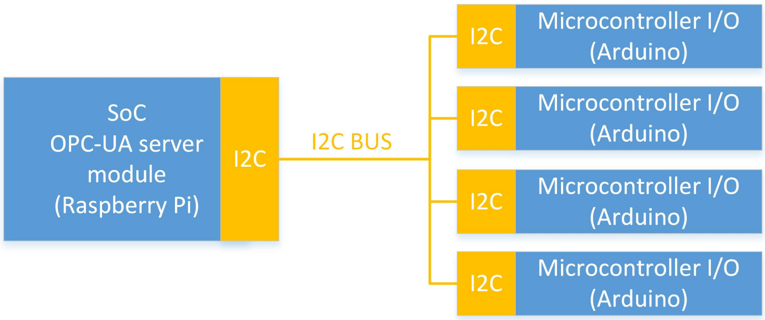


Figure 4.1: Overview of the I^{2}C BUS connections between the SoC hardware (OPC-UA server module) and the microcontrollers (I/O modules). All hardware modules need to support the I^{2}C protocol.

I^{2}C was developed in 1982 and originally designed to connect CPU to peripherals chips in TV sets. It is a multi-master/multi-slave bus that only requires two wires. The two wire signals are SDA (serial data) and SCLK (serial clock). Any number of slaves and masters can be connected to the two lines. Only one device is defined as the Bus Master at a given time while all the others are defined as the Bus Slaves. The Bus Master is defined as the device that initiates the data transfer on the bus. It does that by issuing a "start" condition on the bus, all devices listen on and then the Bus Master sends the "address" of the device it wants to communicate with as well as the read or write command. Then all devices will compare the "address" to their address and simply wait until the Bus Master issues a "stop" condition. The device that has the same address will respond. Only then will the Bus Master start sending the "data" to the slave device. After it has finished sending the data the Bus Master will issue a "stop" condition. Because of the I^{2}C bus ability to listen and write to the wires simultaneously there are no collisions or conflicts. The device trying to connect will simply wait and try again. All I^{2}C devices must have a 7 bit address inbuilt. That results in 128 different I^{2}C devices possible for connection to the Raspberry Pi server, but that is considered more than enough for this prototype [39].

4.1.4 System on chip module for OPC-UA server (Raspberry Pi 2)

Rasperry Pi 2 was chosen as the SoC hardware module that runs the OPC-UA server, (see selection reasoning in Section 4.1.2). It runs on Linux and has support for I^{2}C bus for connection to the microcontroller modules. It connects to a network via Ethernet port but wireless dongles can be bought for little money as an addition. See Figure 4.2 for the Raspberry Pi 2 hardware overview.

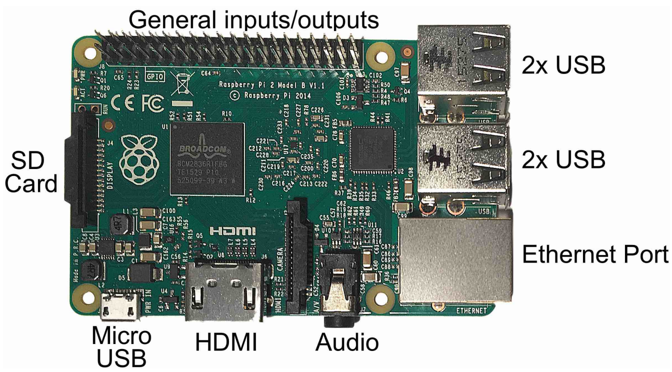


Figure 4.2: Raspberry Pi 2 System on Chip module with Linux. Notes on most important ports used in this project.

The Raspberry Pi foundation has also introduced a new $5 mini version of the Raspberry Pi, named Raspberry Pi Zero. The OPC-UA server can easily be ported over to the Raspberry Pi Zero which makes the form factor smaller and price more appealing. In the midst of this thesis the Raspberry Pi foundation also released the Raspberry Pi 3 which is more powerful than its predecessors and has inbuilt wireless 802.11n and Bluetooth 4.0 capabilities.

Real Time Clock (RTC) addition

A real time clock is not included in the Raspberry Pi hardware and therefore an external one was added. The purpose is to make sure that the actual time and date is known at all times. When Raspberry Pi restarts it loses the date and time settings until it connects to the Internet again. Even though the prototype is aimed at IoT capabilities it is still important that the server knows the time if the Internet access is down after a restart or else the logs will have wrong time stamps until the Internet access is regained again. The Raspberry Pi Foundation justifies the absence of RTC in Raspberry Pi because of added expense in massive production. Mainly since it adds to the size of the board, needs battery and extra components [40]. The real time clock module DS1307 was chosen as it can connect to Raspberry Pi via I^2C [41]. See Figure 4.3 for hardware overview of the RTC module.

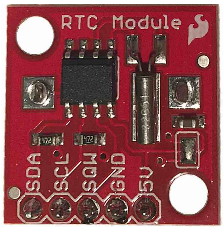


Figure 4.3: Real Time Clock (RTC) of type DS1307 connected to Raspberry Pi via I^2C.

4.1.5 Microcontroller I/O module (Arduino Leonardo)

Arduino Leonardo microcontroller was chosen for all four I/O modules (see selection reasoning in Section 4.1.1). It is designed for I/O applications and has both open source hardware and software. See Figure 4.4 for Arduino hardware overview.

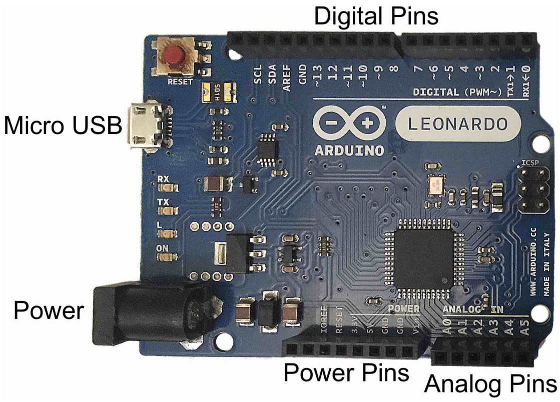


Figure 4.4: Arduino Leonardo microcontroller with notes on most important ports used in this project.

The Arduino hardware is capable of serving all types of signals (DI, DO, AI, AO) in one module. In this project however, only one Arduino is used per signal type, it is required for the modular solution. The total current from all the I/O pins is listed as 200 mA maximum but for any single

I/O pin the maximum is 40 mA [42]. The different I/O settings and capabilities for DI, DO, AI and AO are listed separately in the sections below.

Digital in/out capabilities

The digital pins can both be configured as digital in and digital out, it is defined in the program that is uploaded to the Arduino.

- The digital in I/O module can receive signals, 0 V or 5 V to all four inputs and registers them as LOW or HIGH.
- The digital out I/O module can output signals, either 0 V for LOW or 5 V for HIGH on all four outputs.

Analog in capabilities

The analog in I/O module can receive signal from 0-5 V on all four inputs. It registers the signal through a 10-bit analog to digital converter and maps the input from 0-5 V as integer values between 0 and 1023. The resolution is therefore 5 V / 1024 units or 0.0049 V (4.9 mV) per unit. It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate can be up to 10.000 times per second [43].

Analog out capabilities (additional DAC)

Arduino Leonardo only has Pulse Width Modulation (PWM) instead of "true" Digital to Analog Converter (DAC), see Figure 4.5 for graph of how PWM output dictates the average analog voltage. Arduino Leonardo only has the ability to produce digital signals, either 0 V (LOW) or 5 V (HIGH). It can however change the states between LOW and HIGH very fast, up to frequency of 16 MHz. Duty cycle is the amount of time the signal is on, the state changes happen so fast that the average signal output can be manipulated between 0-5 V with changing the duty cycle. When the duty cycle is, for example, 20% it supplies 1 V and when it is 80% it supplies 4 V. The duty cycle basically dictates a net average voltage output [44].

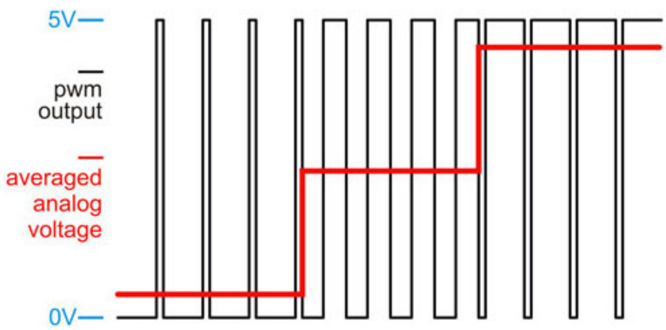


Figure 4.5: Graph of the pulse width modulation showing how the duty cycle dictates the voltage strength [45].

For this project a true analog signal is needed because a lot of actuators use analog signals and will not work with PWM, e.g. proportion control valves. The analog out module therefore needs additional DAC device to be able to output true analog out signal. The additional DAC device, MCP4725, was added to the analog out module and controlled via I^2C. It is low-power, single channel and 12-bit digital to analog converter [46]. In this project it was powered by 12 V power supply that was regulated to 5 V to make sure the power to the DAC was consistent.

Note that the requirement was four analog out ports but there are only two used in this project. It was not because of limitation but because of the hardware availability of the DAC module, only two were available. See Figure 4.6 for overview of the DAC device.

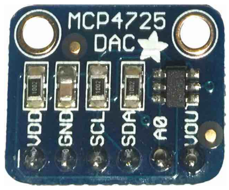


Figure 4.6: Digital to Analog Converter (DAC), connected to and controlled by the Arduino analog out module via I^2C.

4.1.6 Power supply

There are two switching power supplies used for the prototype, one is 5 V DC / 10 W while the other one is 12 V DC / 7 W. They are both connected to 230 V AC. Figure 4.7 shows both power supplies and their connection points. Note that both power supplies have LOW voltage LED indicator which helped confirming that the power needed for each component was enough.

Figure 4.7: Power supply from Carlo Gavazzi in both 10 W/12 V and 7 W/5 V variation. 12 V for Arduino I/O and DAC, 5 V for Raspberry Pi 2 and real time clock.

Power supply (5 V)

The Raspberry Pi is connected to the 5 V power supply. Raspberry Pi then powers the real time clock, two LED's and the I^2C bus. The power requirement for Raspberry Pi is recommended at 5 V and 1.8 A for demanding use, which is 9 W and leaves 1 W left from the 5 V power supply. The low voltage LED indicator has never indicated lack of power even under max load. See Figure 4.8 for the power connection overview.

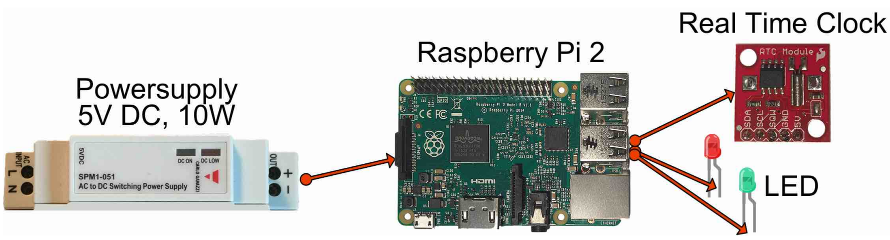


Figure 4.8: 5 V and 10 W power supply from Carlo Gavazzi, powering one Raspberry Pi 2 module and then a real time clock and two LED's powered from the Raspberry Pi.

Power supply (12 V)

All four Arduino modules are powered by the 12 V power supply. The 12 V are also connected to a voltage regulator that maintains 5 V to the DAC modules, see Figure 4.9 for power connection overview. Arduino power requirement is 7-12 V but the ampere needed is relative to the number of I/O used and at what settings. The low voltage LED indicator never indicated lack of power even under max load with all I/O in use and two DAC's at 5 V output.

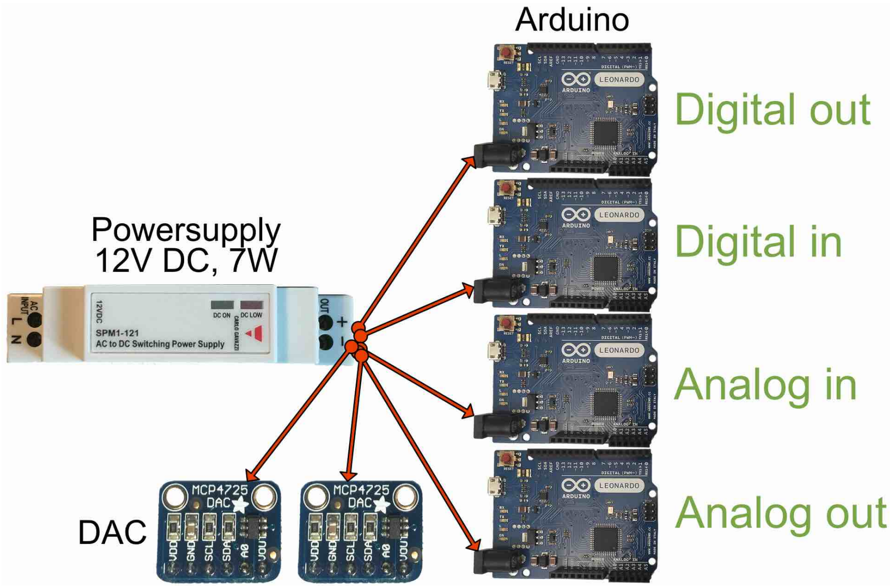
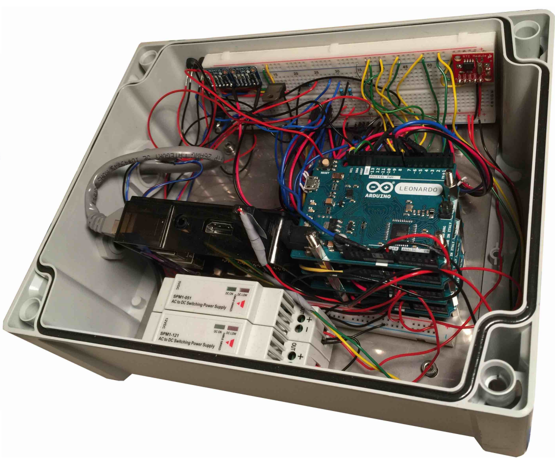


Figure 4.9: 12 V and 7 W power supply from Carlo Gavazzi, powering four Arduino modules and two DAC modules.

4.1.7 Hardware assembly

The final assembly of the prototype can bee seen on Figure 4.10. Table 4.2 lists the pin connections on the I/O modules and is intended for reference with the wiring diagram on Figure 4.11 where all hardware modules and smaller components are shown. Note that all hardware on Figure 4.11 is within the prototype housing. Button one is for shutdown of the Raspberry Pi while button two is for starting the Raspberry Pi up. When initiating shutdown, a red LED indicates that the power off is successful. When restarting the Raspberry Pi, a green LED will indicate when the OPC-UA server is running. A voltage regulator takes in 12 V and regulates it down to 5 V for a stable voltage for the two DAC's. Table

(a) Enclosed prototype



(b) Inside the prototype

Figure 4.10: The final assembly of the prototype in industrial grade housing with all inputs and outputs. (a) shows the enclosed prototype and (b) shows the inside of the prototype

Table 4.2: Pin connections on the Arduino I/O modules and the Raspberry Pi OPC-UA module are listed, including digital in, digital out, analog in and analog out

Arduino Digital in Module	**Pin**	**Arduino Digital out Module**	**Pin**	**Arduino Analog in Module**	**Pin**	**Arduino Analog out Module**	**Pin**	**Raspberry Pi OPC-UA Server Module**	**Pin**
Port 1	4	**Port 1**	4	**Port 1**	A0	**12 V**	Vin	**Shutdown Button**	GPIO 19
Port 2	7	**Port 2**	5	**Port 2**	A1	**I2C bus**	SDA	**Start Button**	SCL
Port 3	8	**Port 3**	6	**Port 3**	A2	**I2C bus**	SCL	**Green LED**	GPIO 5
Port 4	12	**Port 4**	9	**Port 4**	A3	**-**	Ground	**Red LED**	4
12 V	Vin	**12 V**	Vin	**12 V**	Vin			**5 V**	Vin
I2C bus	SDA	**I2C bus**	SDA	**I2C bus**	SDA			**I2C bus**	SDA
I2C bus	SCL	**I2C bus**	SCL	**I2C bus**	SCL			**I2C bus**	SCL
-	Ground	**-**	Ground	**-**	Ground			**-**	Ground

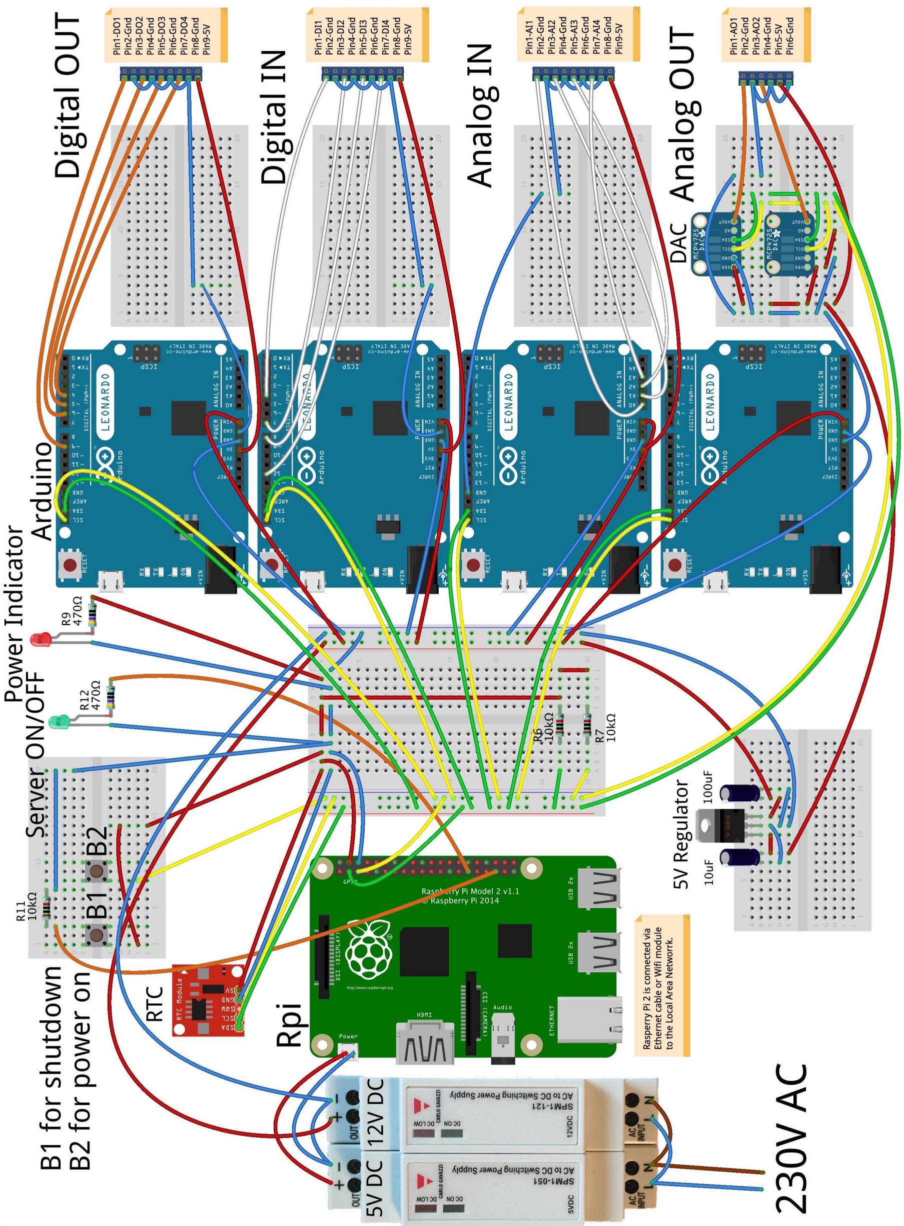


Figure 4.11: Schematic overview of the whole system with all connections, including Raspberry Pi 2 module, four Arduino Leonardo I/O, 5 V and 12 V power supply, real time clock and two DAC's.

4.2 Software

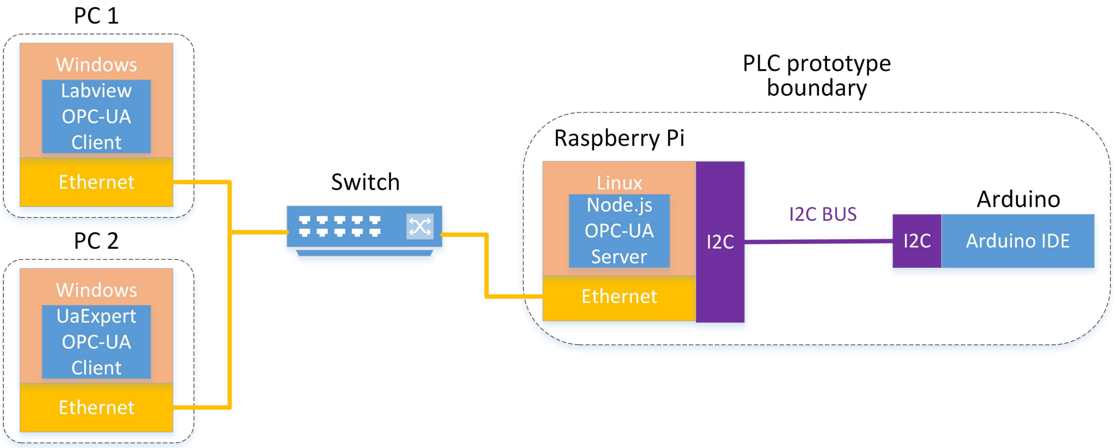


Figure 4.12: Overview of the software used on PC and the prototype. LabVIEW and UaExpert OPC-UA clients were used on PC, Node.js on Linux for Raspberry Pi and Arduino Integrated Development Environment (IDE) on Arduino hardware.

The choice of hardware did narrow the programming tools down, see Figure 4.12 for software overview of the system. Raspberry Pi hardware runs on Linux while Arduino hardware runs on software developed in Arduino IDE based on C/C++. Both Linux and Arduino IDE are known for being open source. Note that both LabVIEW and UaExpert OPC-UA clients used were not open source but were considered the best ones for troubleshooting and testing the device. There is however an open source OPC-UA client available that was briefly tested with the prototype and is being developed within the NodeOPC software project, it is built on javascript and Node.js [13]. It did not possess as powerful tools for testing the prototype as LabVIEW and UaExpert does.

4.2.1 Communication protocol between Raspberry Pi and Arduino modules

For this section it is recommended to observe Figure 4.2 in Section 4.1.3 for overview of the I^2C connections between modules. The communication between Raspberry Pi and Arduino I/O modules occurs on the I^2C bus. Data sent over the I^2C bus is not guarantied to arrive at destination in its original form. Even a single incorrect bit can make the information useless and greatly affect the quality of the data. To help notice errors in the data a checksum error detection was implemented.

The communications between the Raspberry Pi and Arduino I/O modules gets pretty complex because of the amount of information and options it has to include. It was determined that the data string had to include the Source Device ID, Device ID, which port, special command, message length in bytes, message and the checksum. For that reason, a communication protocol was designed that suited this project. The protocol was built with the possibility of expansion in mind for future iterations of the prototype. The message string has length of 8 bytes and a special command byte was also designed into the protocol for expansion benefits. See Figure 4.13 that shows the data string, name and the size of each slot.

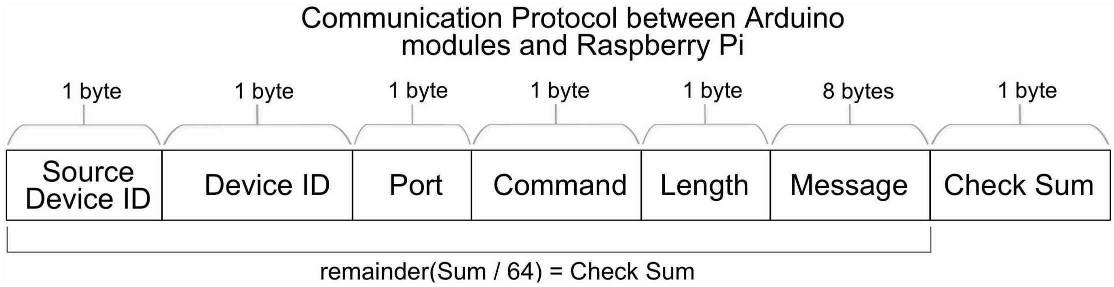


Figure 4.13: The developed communication protocol on the I^2C bus between Raspberry Pi and Arduino I/O modules, showing the protocol details. The remainder of the sum, divided by 64 is the calculated checksum.

The data may have different requirements whether it is received on Arduino I/O end or the Raspberry Pi end. It is nevertheless important to simplify things by having only one universal protocol for both ways.

- **Source Device ID - [1 byte]:** The device sending the data will sign this byte with its ID. The destination module needs to verify the source device ID, both for identification and security.
- **Device ID - [1 byte]:** The ID of the device on the receiving end. It is signed by the device sending the data and can be used by the receiving device to check whether the data was intended for it or not.
- **Port - [1 byte]:** What input / output port is being communicated with, 1-4 for this project.
- **Command - [1 byte]:** This byte offers special commands to be sent to the receiving device. Can be used for expansion of the protocol.
- **Length - [1 byte]:** This is the decimal length of the message, for easier data manipulation in the I/O modules.
- **Message - [8 bytes]:** The message has length of 8 bytes. Note that it is not used in the fashion of High/Low bytes. For now, each byte is supposed to be used as one integer from 0-9 when sending a number message, that means that it can transmit and receive integer between 0 - 99.999.999. The High/Low byte could be considered more efficient but at this stage of the prototype it is more important that the message is simpler for troubleshooting purposes.
- **Checksum - [1 byte]:** It is the remainder of the sum of all other bytes converted to ASCII. The destination device will then calculate the checksum again for all bytes except the checksum byte and compare to the original checksum byte.

The checksum is calculated on the sender's end and then calculated again at the receiving end and compared. If they are not exactly the same then the last valid value is used.

4.2.2 Raspberry Pi

See Section 4.1.1 for the reasoning of choosing Raspberry Pi as an OPC-UA server module and Section 4.1.4 for overview of the Raspberry Pi hardware. By choosing the Raspberry Pi hardware it narrowed the choice of operating system to Linux or Windows 10 IoT core. Linux is open source and was therefore chosen as the operating system for Raspberry Pi OPC-UA server. In next sections the settings and software running on Raspberry Pi will be listed and explained.

For reference in next sections see Figure 4.14 for a simple flow diagram showing the functionality of the Raspberry Pi from start-up and until the OPC-UA server has begun running.

When the Raspberry Pi is powered up, it boots up to Linux and then gets the date and time from a hardware real time clock. Crontab is a system daemon which is similar to Windows Services and is configure to run at Linux startup. Crontab then starts up a Command Line Interface (CLI) tool called Forever. Forever then starts up the OPC-UA server and constantly checks if it is running or not. If the OPC-UA server is not running, then Forever starts it up again. That way it is made sure that the OPC-UA server is always running.

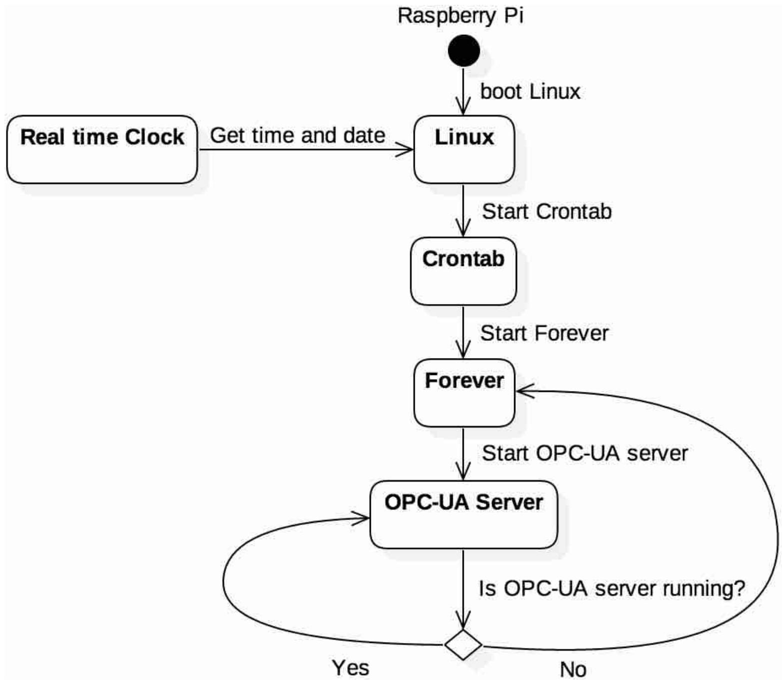


Figure 4.14: Simple flow diagram of the software functionality on the Linux operating system on Raspberry Pi. It shows how it uses the installed Crontab to automatically start Forever that runs the OPC-UA server and keeps it running, even in the case of server crash.

Real time clock (RTC)

See Section 4.1.4 for the hardware overview of the RTC and reasoning for its addition. To set up the DS1307 external RTC it is necessary to enable the I^2C bus in the system settings in Linux. Adafruit has a tutorial that was used to set the Raspberry Pi to read from the added RTC [47]. Now instead of the Raspberry Pi looking to the web for date/time reference it will set it self to the real time clock that was added.

Crontab start up services

Even though the Forever tool keeps the server script alive at all times it does not automatically run at start-up when the Raspberry Pi restarts. The Crontab is a system deamon, that is similar to Windows Services but on Linux. It includes a text file with a list of commands meant to be run at specific times [48]. Crontab was used in this project to include the Forever CLI tool at start-up. When Forever CLI starts it automatically runs the OPC-UA server and keeps it running.

Forever Command Line Interface (CLI) tool

Forever is a simple CLI tool for ensuring that a given script runs continuously (forever). There are two ways of using the tool: Using it programmatically or use it by itself through the command line [49]. In this project it is used through the command line and it makes sure that the OPC-UA server script is always running, in case of a software crash it will start the script up again.

OPC-UA server

OPC-UA is an open source specification that enables software developers to develop OPC-UA applications. Those applications can then be verified by the OPC Foundation, it is a quality stamp that makes sure that all the specifications were correctly applied. While OPC-UA is an open source specification it's applications are not necessary free nor open source.

There are a lot of software developers and vendors that are developing and selling OPC-UA software but there are open source developments of OPC-UA available. One such was chosen for this project, it is named NodeOPCUA and is an OPC-UA stack fully written in javascript and Node.js [13]. Note that a protocol stack is a set of network protocol layers that work together, based on the OSI model. The asynchronous nature of the Node.js programming language is considered a strong point when considering the communication speed. It makes it a good choice for server based applications.

The OPC-UA code can be seen in Code A.1 in Appendix A.3.1. It has 17 variables, for all I/O ports and information from the performance of the Raspberry Pi. The information from Raspberry Pi is gathered to monitor the performance of the processor, memory usage and the overall runtime while the OPC-UA server is running. It can be useful to see how the Raspberry Pi hardware handles the OPC-UA server on full load.

Variables used in the OPC-UA server:

- Digital in 1-4 (four variables for each port)
- Digital out 1-4 (four variables for each port)
- Analog in 1-4 (four variables for each port)
- Analog out 1-2 (two variables for each port)
- Raspberry Pi up time in hours
- Raspberry Pi CPU average load for last 15min
- Raspberry Pi CPU Percent of memory used

The OPC-UA server needs to communicate to the General Purpose Input Output (GPIO) pins on the Raspberry Pi for turning LED's ON/OFF for "OPC-server ON" and "successful shutdown." It also has to be able to communicate with the I/O modules over the I^2C bus, see Figure 4.11 in Section 4.1.7 for the wiring overview. The following libraries were used in Node.js for the OPC-UA server, all installed through the npm package manager for JavaScript [50].

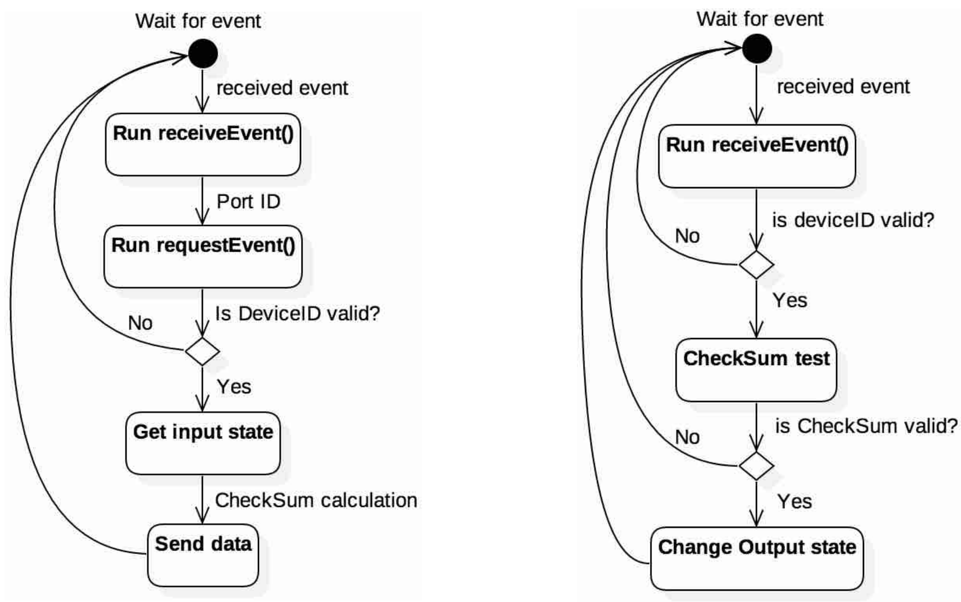
- **"node-opcua"** is the OPC-UA library needed for the server [51].
- **"os"** library is for accessing the CPU, Memory and Run time information from Raspberry Pi [52].
- **"i2c"** is the Library for communication over the I^2C bus [53].

- **"sys"** library is for directly accessing the command line for shutting down the Raspberry Pi.

- **"onoff"** library is for accessing the GPIO pins on the Raspberry Pi for indication of the red LED for power and the green LED for server running [54].

4.2.3 Arduino

See Section 4.1.2 for the reasoning of choosing Arduino as a microcontroller and Section 4.1.5 for overview of the Arduino hardware. The programming language for Arduino is called Arduino IDE, it is merely a subset of the C/C++ language [55]. The I/O modules are responsible for reading sensors and control actuators. They receive or send data through I^{2}C bus to Raspberry Pi. The programming differs in some way between all I/O modules but the digital in and analog in share the same structure as well as digital out and analog out. See Figure 4.15 for flow diagram of both programming procedures for reference in next sections. Note that the event functionality is built into the I^{2}C library



(a) Digital in and analog in

(b) Digital out and analog out

Figure 4.15: Flow diagram for the Arduino communicating with OPC-UA server. (a) shows Arduino collecting sensor data and sending it to OPC-UA server. (b) shows Arduino receiving data from OPC-UA server and outputs to actuators.

Digital in

The digital in module reads the input ports, either LOW or HIGH (0 or 1). It only reads the ports when it receives a request from the OPC-UA server with byte indicating port ID. That means that the Arduino is not constantly reading the input ports unless it is told to do so, that saves power and resources in the microcontroller. When it receives a request from the OPC-UA server it checks if the DeviceID is valid, reads the specific input port and calculates the checksum error. It then builds the custom protocol string as described in Section 4.2.1 and sends the data to OPC-UA server. See Code A.2 for reference in Appendix A.3.2.

Analog in

The analog in module has the same functionality as the digital in module but instead of registering only LOW/HIGH (0 or 1) it registers the Analog to Digital Converter (ADC) value between 0-1023. See Code A.3 for reference in Appendix A.3.3.

Digital out

The digital out module delivers signals, either 0 V for LOW or 5 V for HIGH. It only changes the output signal when the OPC-UA server requests the state change for the specific port. The Arduino receives the data string from the OPC-UA server and reads the information. There is a device ID check in the beginning of the code and it does not allow any changes until the right device ID is confirmed. It calculates the checksum error and compares it to the checksum value sent from the OPC-UA server. If the checksum error value and the calculations are equal it changes the state of the output pin as requested. See Code A.4 for reference in Appendix A.3.4.

Analog out

The analog out module has the same functionality as the digital out module but instead of outputting only LOW/HIGH signal it outputs analog signal through external DAC device. The DAC is 12 bit which means that the value is controlled from the integer range of 0-4095 which then correspond to 0-5 V. An external library is needed for control and is available from the DAC manufacturer [56]. See Code A.5 for reference in Appendix A.3.5.

Measurement Arduino

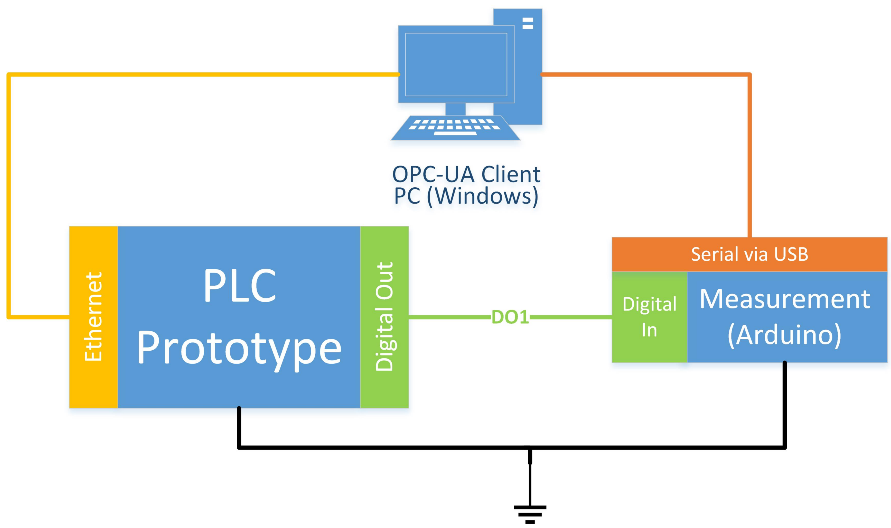


Figure 4.16: Overview of the connection setup of the measurement Arduino when measuring the communication speed. It measures the time between state changes from the digital output from the prototype. The timing information from the measurement Arduino are logged down through terminal program via Universal Serial Bus (USB) on a PC.

It was important for the requirements to measure the exact read and write communication speed from OPC-UA client to sensors and actuators. To be able to fulfill that requirement it was necessary to use an additional Arduino which has the sole purpose to measure the communication speed, see connections overview on Figure 4.16. The OPC-UA client on the PC is configured to change states as fast as possible on a single digital out port. The measurement Arduino then times the communication speed on that specific port. It measures the exact time it takes for the

OPC-UA digital output to change its pin state from LOW to HIGH. That way it was possible to measure the timing from a client to the actuator, see simple flow diagram on Figure 4.17. See the Code A.6 for reference in Appendix A.3.6.

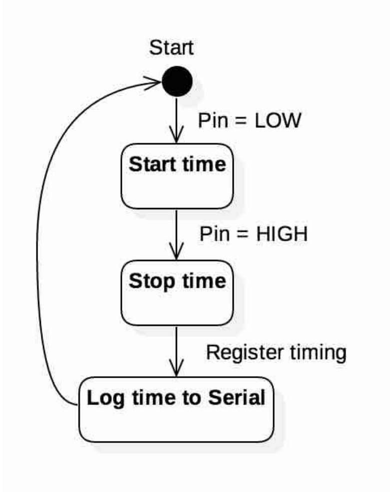


Figure 4.17: Simple flow diagram of the Arduino measurement programming. It starts timing when the pin is registered as LOW (0 V) and stops the time when it is changed to HIGH (5 V), then it registers the timing value to serial. See the Code A.6 for reference in Appendix A.3.6.

The measurement Arduino was also used to calculate and log the frequency of the checksum error from OPC-UA server, over to digital out and analog out modules through the I^2C bus, see Figure 4.18 for the connection overview. Note that Figure 4.18 only shows the measurement setup for the digital out module, the exact same setup was used for the analog out module. The checksum is calculated and sent with the data string trough the I^2C bus to the digital out module. The checksum error is calculated again at the digital out module and both of them are compared. If they are not equal it means that there is a checksum error detected and then a short pulse is sent to the measurement Arduino. The measurement Arduino will then register the pulse and send the information to the OPC-UA client.

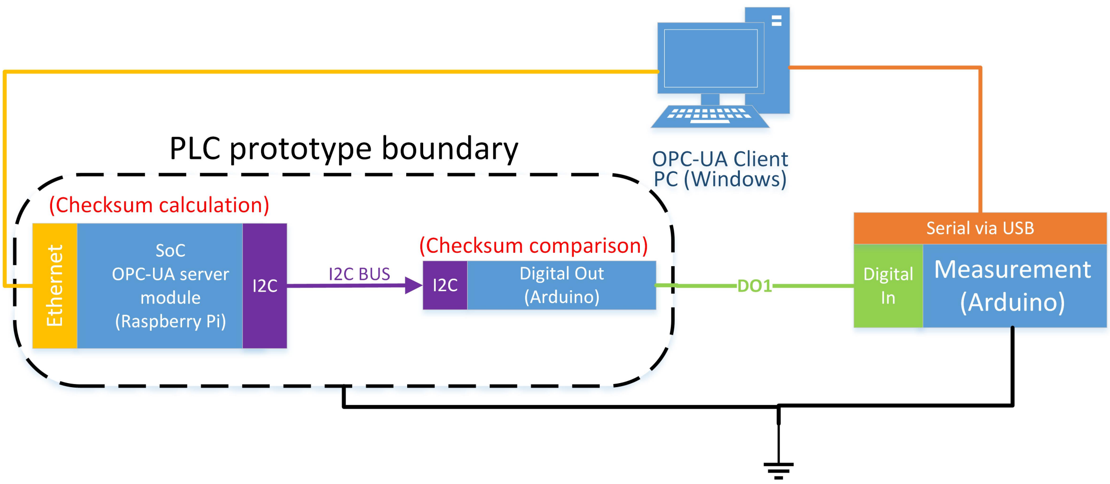


Figure 4.18: Overview of the connection setup of the measurement Arduino when it measures the frequency of checksum error. When checksum error is detected the digital out module sends out a pulse. The timing information from the measurement Arduino are then logged down through USB on a PC.

Both the digital out and analog out programs had to be modified to output a HIGH signal to a selected pin for 50 ms (pulse) whenever there was a checksum error. That way it was possible to register the checksum error over the I^2C bus without using the serial on the I/O modules. The reason for not simply using the serial on the I/O modules to log the checksum errors was that whenever the I/O modules were connected via USB it interfered and resulted in excessive checksum errors. Possible reason for this behavior was that while the Arduino I/O module is connected to PC it receives 5 V that in some way interferes with the I^2C bus. See the modified analog out Code A.7 for reference in Appendix A.3.7. See the modified digital out Code A.8 for reference in Appendix A.3.8

4.2.4 LabVIEW OPC-UA client

Overview of the LabVIEW OPC-UA client connection to the prototype can be seen on Figure 2.2 in Section 2. To be able to do experiments with the system it was necessary to have an OPC-UA client that is highly customisable. An OPC-UA client was therefore developed in LabVIEW on a Windows PC. It was configured to access all 17 variables defined in the OPC-UA server. It was mainly used for reading and writing to the OPC-UA server to verify that all I/O modules were working properly on all input and output ports. The client was implemented with a logging option that was used for logging all variables. The LabVIEW client has the option of adjusting how often per second it would read/write the data from the OPC-UA server, that was done for experimenting with how fast it could read/write the data. Setting the loop to the fastest settings made the processing power of the OPC-UA server the bottleneck in the system and therefore was a good way to test it at full load. See the LabVIEW code on Figure A.1 for reference in Appendix A.3.10. See the front-panel Graphical User Interface (GUI) on Figure A.2 for reference in Appendix A.3.10.

Chapter 5

Prototype test plan

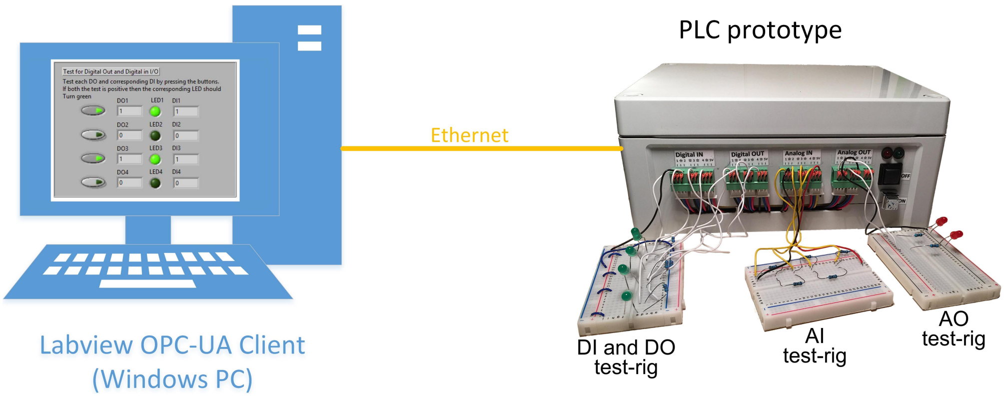


Figure 5.1: Overview of the Programmable Logic Controller (PLC) prototype with all three test-rigs presented and connection to a Personal Computer (PC) running Open Platform Communications Unified Architecture (OPC-UA) client via Ethernet.

This section is about the procedure of testing the prototype, see Figure 5.1 for an overview of the actual prototype with all test rigs connected to it and connection to the LabVIEW OPC-UA client. Note that "test-rig" in this context means the physical breadboards and components used plug into the prototype for testing. A printable version of the test plan, called a "test plan document" can be seen in Appendix A.3.10. It is a structured collection from this chapter and intended as instructions for a technician to carry out the tests and note down the results. The prototype schematics can be seen in Figure 4.11 in Section 4.1.7. The tests are intended to make sure that all four types of input and output modules (DI, DO, AI, AO) are working correctly. These tests can be programmed through a OPC-UA client of choice but in this thesis a OPC-UA client in LabVIEW was used. A test plan document can be seen in Appendix A.3.10, it is a printable version of the test plan addressed in this chapter.

5.1 Digital in and digital out

Both the digital in module and digital out module are tested together. Digital in module is tested with the outputs from digital out module. The software flow diagram can be seen on Figure 5.2 where the testing of one input port and one output port is shown. The functionality is then extended for four input and output ports.

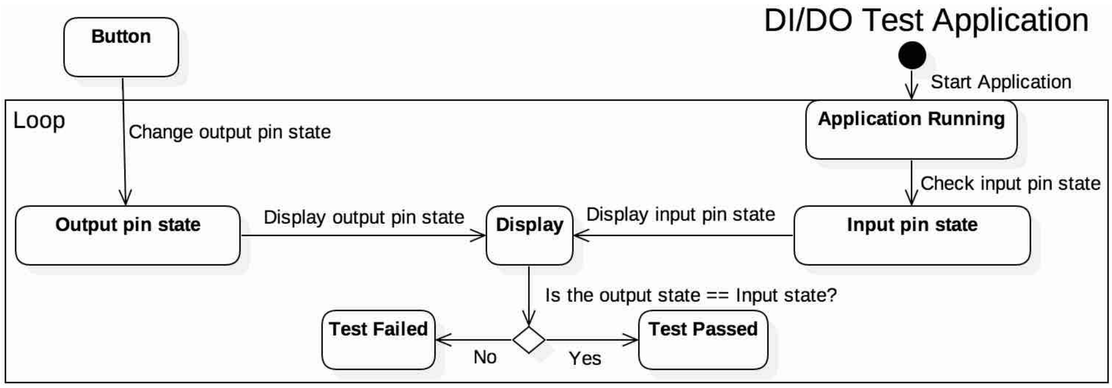
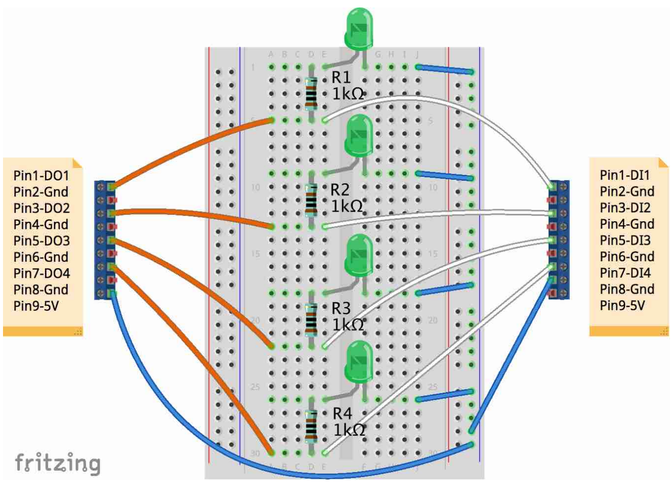
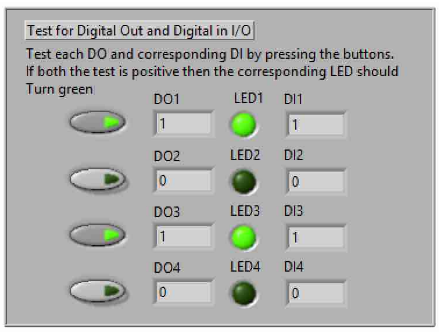


Figure 5.2: Simple flow diagram of the software that is responsible for testing digital in and digital out modules. It shows state change for one output pin and one input pin which is compared on the display. The program can then be extended for four pins.

The test-rig and test-program Graphical User Interface (GUI) can be seen on Figure 5.3. The four input ports are tested with the output ports by pressing a corresponding button on the LabVIEW test program. If the digital output port is working correctly, then corresponding digital in port will respond with a green LED. If, for example, DO1 is LOW (0) and then turned to HIGH (1) then DI1 port should indicate the value 1 with a green LED. If DI1 will not respond to a change in DO1 then the test failed.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 5.3: Test-rig where digital in ports are tested with digital out ports in LabVIEW. HIGH and LOW signals are outputted and checked whether corresponding signals appear on the inputs.

5.2 Analog in

The software flow diagram can be seen on Figure 5.4 where ADC value is measured and converted to voltages. The specific voltage is then compared to predetermined range that is considered acceptable.

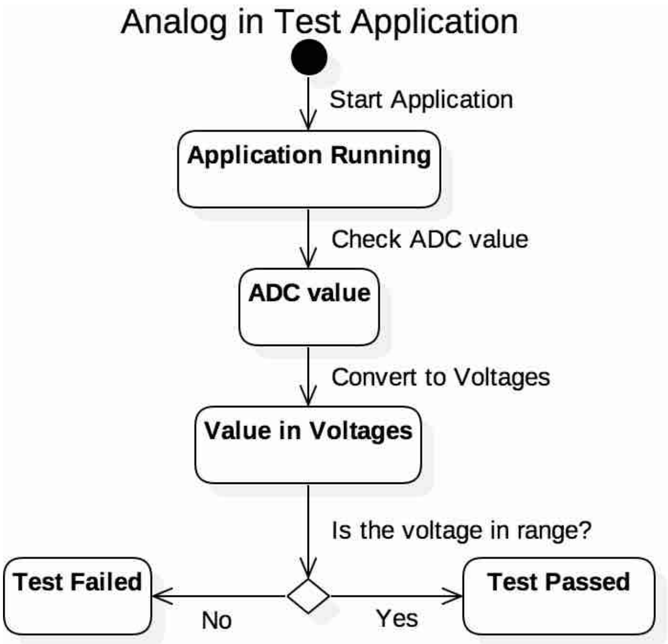
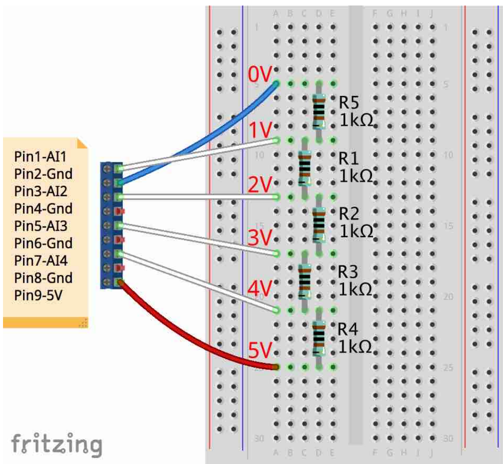
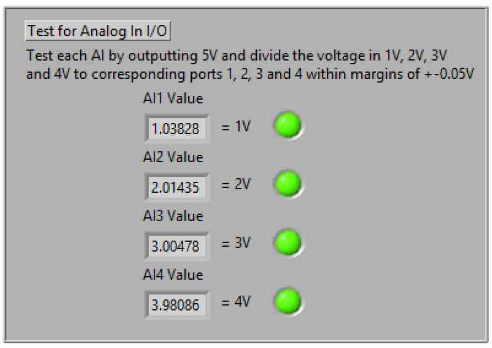


Figure 5.4: Simple flow diagram of the software that is responsible for testing AI modules. It acquires the ADC value, converts it to voltage and compares it to the specific, allowed range. This program is then extended to four ports.

The test-trig and test-program GUI can be seen on on Figure 5.5. The test-rig has 5 V supplied that is divided with four 1 kΩ resistors which results in 1 V, 2 V, 3 V and 4 V to the corresponding ports 1, 2, 3 and 4. The LabVIEW program approves with a green LED if the readings are within 0.05 V from the correct values, else the test fails.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 5.5: Test-rig for making sure the analog in module is reading the correct voltages, it reads 4 V, 3 V, 2 V and 1 V on the relative ports.

5.3 Analog out

The software flow diagram can be seen on Figure 5.6 where user sets specific voltage to a DAC. The voltage is then measured with a multimeter and compared to predetermined range that is considered acceptable.

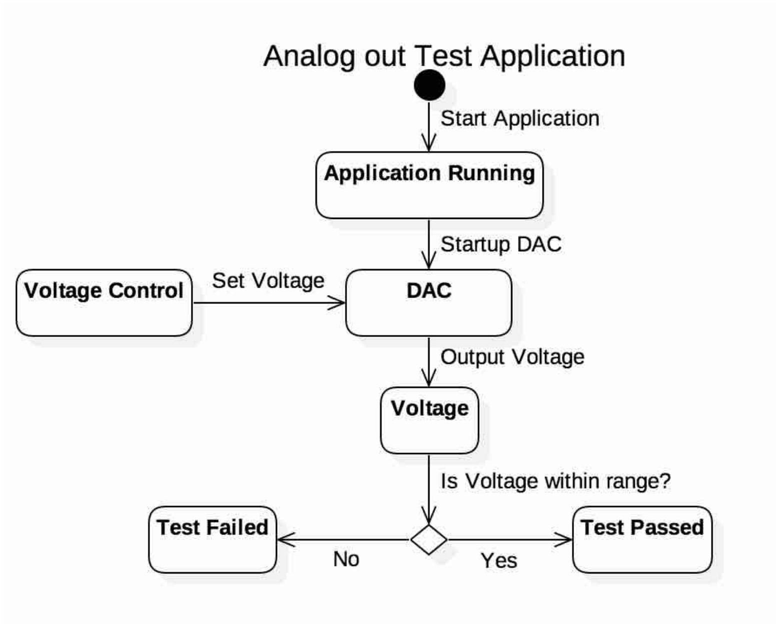
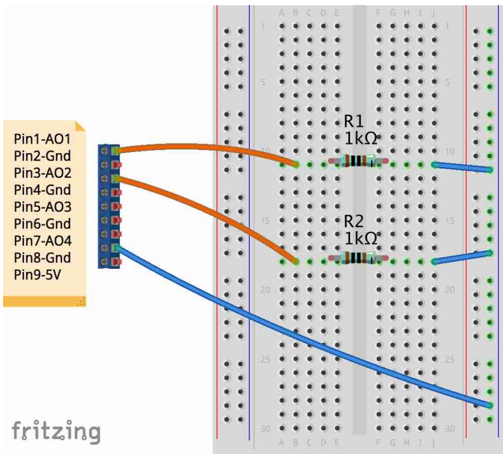
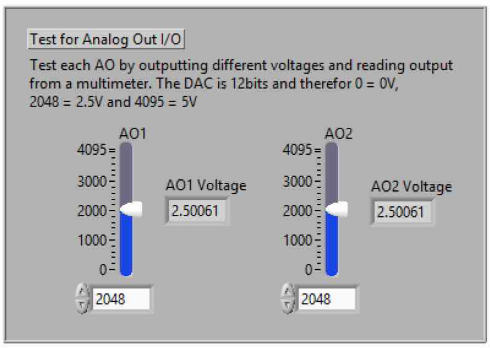


Figure 5.6: Simple flow diagram of the software that is responsible for testing AO modules. The user sets the output voltage and then measures the output with a multimeter and compare.

The test-rig and test-program GUI can be seen on Figure 5.7. It makes sure that the analog out module is working properly. The multimeter is needed to measure the voltage drop over the 1 KΩ resistors and compare it to the output command in the LabVIEW test program.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 5.7: Test-rig for making sure the analog out module is outputting the correct voltages. The multimeter is used to check the voltage drop over the resistors and then compared to the controlled output.

5.4 Test plan check list

On Table 5.1 is the check list that is supposed to be complementary to the test applications for a technician to fill out when testing the prototype. It was filled out for demonstration purposes by the author but an unfilled one is available with the test plan document in Appendix A.3.10. The LabVIEW test client is used for testing all I/O modules by giving commands to outputs or reading the inputs. Digital out ports are supposed to be changed from LOW to HIGH and it should then register both on the green LED's for each output and on values in the GUI. Digital in ports should be monitored at the same time as digital out and it should register the changed state from LOW to HIGH. Analog out should be measured with a multimeter while the output is changed from 0 V, 2.5 V and 5 V. Analog in is supposed to read 1 V, 2 V, 3 V and 4 V on corresponding ports. The check list is supposed to be filled out "Passed" or "Not Passed" to indicate whether the test failed or passed.

Table 5.1: Prototype test plan check list that is used with test applications. It is intended for a technician to fill in as a check list. This particular list was filled out for demonstration purposes by the author. The unfilled check list is available in the test plan document in Appendix A.3.10.

Modules	**Ports**	**Description**	**Passed**	**Not Passed**	**Comments**
Digital out		**Set ports output from LOW to HIGH**			
-	1	Output should change states	✓		
-	2	-	✓		
-	3	-	✓		
-	4	-	✓		
Digital in		**Register states from digital out**			
-	1	Input should change states	✓		
-	2	-	✓		
-	3	-	✓		
-	4	-	✓		
Analog out		**Set voltage to 0 V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$	✓		
-	2	-	✓		
-	3	-		✓	No DAC on port 3
-	4	-		✓	No DAC on port 4
-		**Set voltage to 2.5 V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$	✓		
-	2	-	✓		
-	3	-		✓	No DAC on port 3
-	4	-		✓	No DAC on port 4
-		**Set voltage to 5 V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$	✓		
-	2	-	✓		
-	3	-		✓	No DAC on port 3
-	4	-		✓	No DAC on port 4
Analog in		**Set up the analog in test-rig with 5 V supply**			
-	1	Should Register 1V within $\pm 0.05V$	✓		
-	2	Should Register 2V within $\pm 0.05V$	✓		
-	3	Should Register 3V within $\pm 0.05V$	✓		
-	4	Should Register 4V within $\pm 0.05V$	✓		

Chapter 6

Results and discussion

6.1 Results overview

This section lists a short overview of content in the results chapter. Sections are indicated with bold text while contents are enumerated in correct order. First four sections are directly linked to the requirements while the section "additional results" is not.

Communication speed performance
The communication speed is measured at:

1. OPC-UA client to OPC-UA server with different encryption and message signing settings (read and write)
2. Round time from OPC-UA client to actuator and then back from sensor to OPC-UA client
3. From sensor to OPC-UA client (read)
4. From OPC-UA client to actuator (write)

Communication reliability

1. The amount of checksum error from Arduino I/O module to Raspberry Pi OPC-UA server on the I^2C bus (digital in and analog in)
2. The amount of checksum error from Raspberry Pi OPC-UA server to Arduino I/O module on the I^2C bus (digital out and analog out)

Hardware reliability

1. Prototype reliability at long term run
2. Prototype reliability at power cut-off

Other results

1. Choice of hardware and software
2. Open source status of hardware and software
3. Status of the prototype ability for standalone capabilities (buttons, services, LED)
4. Modular design and I/O modules types
5. OPC-UA communication integration and custom communication protocol on I^2C bus

Additional results

1. Raspberry Pi performance monitoring (CPU and memory)
2. Unsuspected behavior on analog in ports

6.2 Communication speed performance

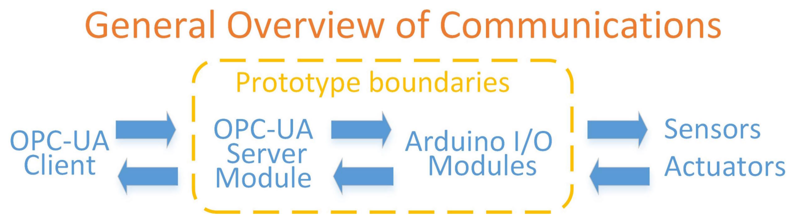


Figure 6.1: Overview of the communications, blue arrows indicate the communication way and the purple box shows the prototype boundaries.

On Figure 6.1 is the overview of the communication between the hardware components. It is important to understand the communication overview before continuing to next sections as it will be used for clarification in the sections to come. The blue arrows point to the direction of communication and are later marked green to indicate which communication way is being used. The hardware that is communicating with each other is also marked green for clarification. The hardware responsible of collecting the data is indicated with a purple box.

6.2.1 OPC-UA server to OPC-UA client communication speed

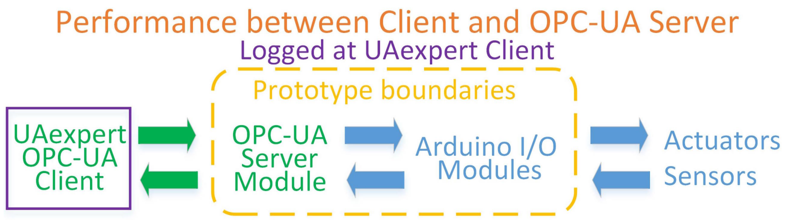


Figure 6.2: Overview of the measurement setup for the communication between the UaExpert client and OPC-UA server.

The communication between client and server offer a range of security settings which include message security mode and security policy. The communication speed results from different kinds of encryption and message signing will help to understand the amount of drawback in longer communication time for the added security. On Figure 6.2 is an overview of the measurement setup where the client in this case is the UaExpert from Unified Automation. It times the read and write communications between OPC-UA server and client with a OPC UA performance plugin.

UaExpert client was configured to read and write to OPC-UA server as fast as it could for 10 minutes, setting the sampling interval to zero. That means that the speed of communication starts to depend on the hardware processing power used for the OPC-UA server module. It logged down the milliseconds it took to read and write each variable. The measurements results can be seen in Table 6.1 where different settings of encryption and message signing result in higher communication time. With no security policy and no message security mode the read speed was on average 6.12 ms while the write speed was slightly lower with average of 6.38 ms. Adding signed communication resulted in 11-12% slower speed for both read and write. Adding signed message mode with Basic128Rsa15 encryption resulted in 56-57% slower speed for both read and write. Adding signed message mode with Basic256 encryption resulted in 57-58% slower speed for both read and write. See Table 6.2 the added time in milliseconds for each settings variation on both read and write.

Table 6.1: Communication performance from UaExpert client to OPC-UA server with comparison between different signing and encryption settings.

Security Policy / Message Security Mode	Read [ms]	Write [ms]
None / None	6.12	6.38
Basic128Rsa15 / Signed	6.84	7.1
Basic256 / Signed	6.81	7.14
Basic128Rsa15 / Signed&Encryption	9.57	9.99
Basic256 / Signed&Encrypted	9.63	10.02

Table 6.2: The amount of milliseconds per settings that are added to the communications.

Write	**Basic128Rsa15 / Signed**	**Basic128Rsa15 / Signed&Encryption**	**Basic256 / Signed**	**Basic256 / Signed&Encrypted**
Added time to 6.12 ms	0.72 ms	3.45 ms	0.69 ms	3.51 ms
Read	**Basic128Rsa15 / Signed**	**Basic128Rsa15 / Signed&Encryption**	**Basic256 / Signed**	**Basic256 / Signed&Encrypted**
Added time to 6.38 ms	0.72 ms	3.61 ms	0.76 ms	3.62 ms

Discussion

Prosys presented OPC-UA performance results measured on an older version of Raspberry Pi which is about twice as slow as the newer Raspberry Pi 2 used in this thesis. The results from Prosys will be compared to the results in this thesis for comparison. Prosys measured that adding encryption to a method call resulted in time addition of about 3.9 ms which was little bit higher than the average of 3.53 ms measured in the prototype [57]. That was expected because of the faster hardware on the Raspberry Pi 2. Note that details of the Prosys experiment setup is not fully known and therefore this comparison is to be taken as such. The requirement of communication speed under 100 ms between OPC-UA client and OPC-UA server module was met and resulted in average of 9.63 ms for read and 10.02 ms for write.

6.2.2 Round trip communication speed

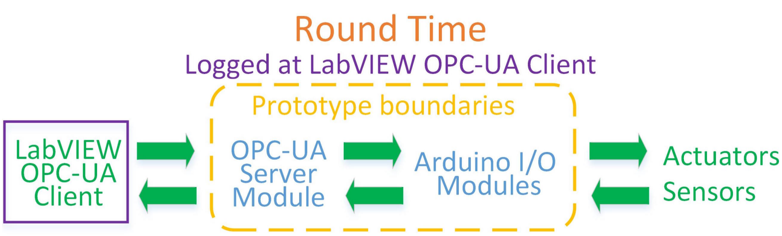


Figure 6.3: Overview of the measurement setup for the communication from LabVIEW client out and back in to the same LabVIEW client.

The round trip communication speed was measured by a LabVIEW client. On Figure 6.3 is the measurement setup with green arrows showing the communication way which begins and ends with the LabVIEW client. The LabVIEW client sends out a signal to DO/AO, it is received in DI/AI and travels back to the LabVIEW client. The round trip was then timed in the LabVIEW client while trying to write and read as fast as possible. The measurement setup was running for about 20 min and the data was then analyzed in Matlab. The OPC-UA security settings were Basic128Rsa15 / Not Signed. The round-trip resulted in average of 19.14 ms for digital out to digital in. The round-trip for analog out to analog in resulted in slightly slower time of 20.00 ms.

6.2.3 Write time communication speed

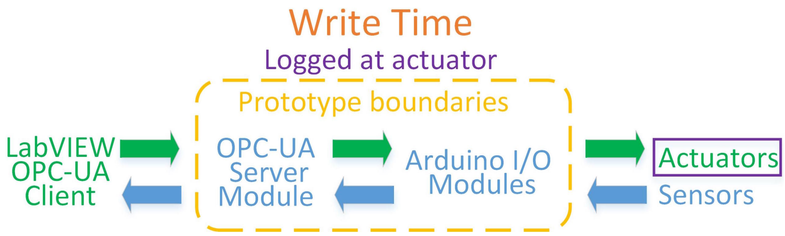


Figure 6.4: Overview of the measurement setup for the communication from LabVIEW client out to a measurement Arduino.

The total time from the LabVIEW client to an actuator was measured with the measurement Arduino, described in Section 4.2.3. It was used to time the pin state change from HIGH (5 V) to LOW (0 V). On Figure 6.4 is the measurement setup with the green arrows showing the communication way which begins at the LabVIEW client and ends with the measurement Arduino. The measurement setup was running for about 20 minutes with security settings Basic128Rsa15 / Not Signed. It resulted in a measured write time of 7.89 ms for analog out and slightly slower write speed of 8.13 ms with digital out.

6.2.4 Read time communication speed

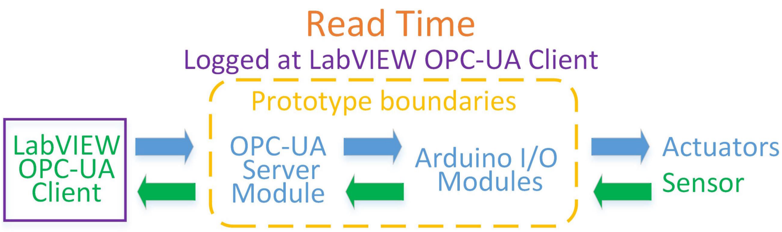


Figure 6.5: Overview of the read time communication way from Sensor to client.

The time was known from the round-trip and the write time to the actuator. That knowledge was used to calculate the time for the read time. On Figure 6.5 is the setup with the green arrows showing the communication way which begins at the sensor and ends with the LabVIEW client. The read time for digital in resulted in 11.01 ms while the analog in resulted in slightly higher communication speed of 12.11 ms.

6.2.5 Comparison

The communication comparison was collected and interpreted in Table 6.3. It shows that the write time is faster than the read time and digital is slightly faster at both writing and reading than the analog.

Table 6.3: Communication performance from LabVIEW client

Module	**Round Trip [ms]**	**LabVIEW to Module Output (Write time) [ms]**	**Module Input to LabVIEW (Read time) [ms]**
Digital	19.14	8.13	11.01
Analog	20	7.89	12.11
Average	19.57	8.01	11.56

Another test was done through the LabVIEW client where read calls of all 17 variables defined in the OPC-UA server were plotted over a time span of 18 hours. They were read at maximum speed with full load on the system. For 311.862 sample points the average call was 11.89 ms while the max call time was 45.92 ms, see Figure 6.6 for plotted results.

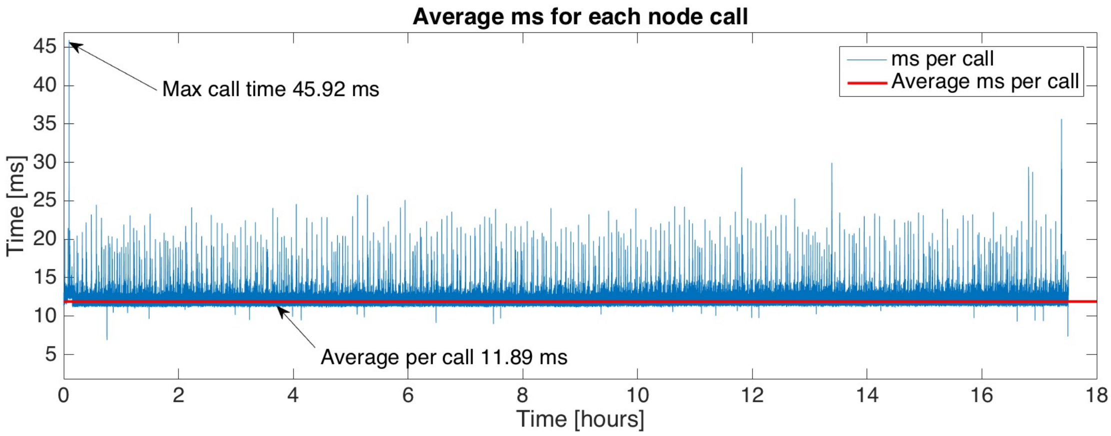


Figure 6.6: Graph of 311.862 sample points over the time-span of 18 hours. This was logged in the LabVIEW OPC-UA client in a full load on the system where the data was read/written as fast as possible.

6.2.6 Discussion

The round time measured from the OPC-UA performance evaluation from Prosys was 20 ms with no security and no message signing and about 31 ms with security settings Basic128Rsa15 / Not Signed [57]. Prosys used an older Raspberry Pi platform which has about half the power of the Raspberry Pi 2 used in the thesis. When comparing the results of the 19.57 ms average round trip time in the prototype it is clear that the extra processing power in Raspberry Pi 2 shortens the communication speed. Especially considering the 19.57 ms round trip in the prototype has the security settings Basic128Rsa15 / Not Signed which is more comparable with the 31 ms round trip measurement observed from Prosys. Note that details of the Prosys experiment setup is not fully known and therefore this comparison is to be taken as such.

The read time was on average 11.56 ms while the write time was 8.01 ms. When plotting the read time from 17 variables on full load it resulted in 11.89 ms which is little more than the 11.56 ms when only reading one variable as fast as possible. The requirement for communication speed of under 200 ms for roundtrip and under 100 ms for write was met and was about 8-10 times faster than the required speed.

6.3 Communication reliability

To make sure that the communication is reliable from the OPC-UA server to its I/O modules it is important to measure the error percentage from the checksum error detection.

6.3.1 Digital in and analog in

The digital in and analog in readings come from the Arduino modules to the OPC-UA server. That means that the checksum detection and error logging is done in the OPC-UA server. See Figure 6.7 for an overview of the measurement setup.

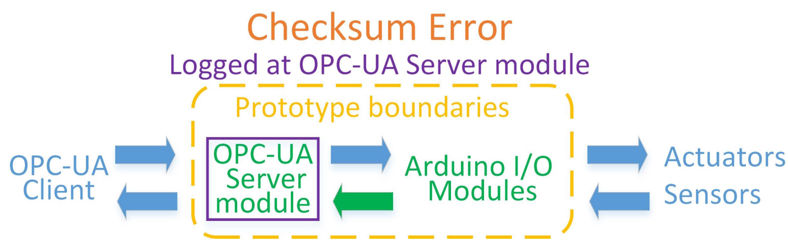


Figure 6.7: Overview of the measurement setup for the checksum error from Arduino module to OPC-UA server.

The Raspberry Pi logged for 28.5 hours to a text file whenever error was detected. The average checksum error was 0.054%. Error percentage for all ports can be seen on Table 6.4.

Table 6.4: Total DI and AI samples and checksum errors collected from the OPC-UA server.

I/O	Port	Total Samples	Total Errors	Error Percentage [%]
Analog In	1	641145	337	0.053
-	2	646498	376	0.058
-	3	646531	304	0.047
-	4	644718	325	0.050
Digital In	1	643167	354	0.055
-	2	646423	378	0.058
-	3	642281	354	0.055
-	4	645605	344	0.053
	Sum	5156368	2772	
			Average	0.054 %

It was interesting to see whether the errors were occurring at the same time at all ports due to a global interference or only at one port at a time. Global interference from unforeseen factors could be troubleshooted through a data analysis and comparison between error logs.

On Figure 6.8 is a graphical representation of errors from all four ports of AI collected for over 10 hours. The errors are plotted in percentages since last error was detected, e.g. if two errors occur in a row then the error percentage is 100%. There are four subplots for each analog input pin and when they are compared to each other a pattern between errors can be observed. It does not seem to show grouping of errors and therefore implies that the error is randomly distributed.

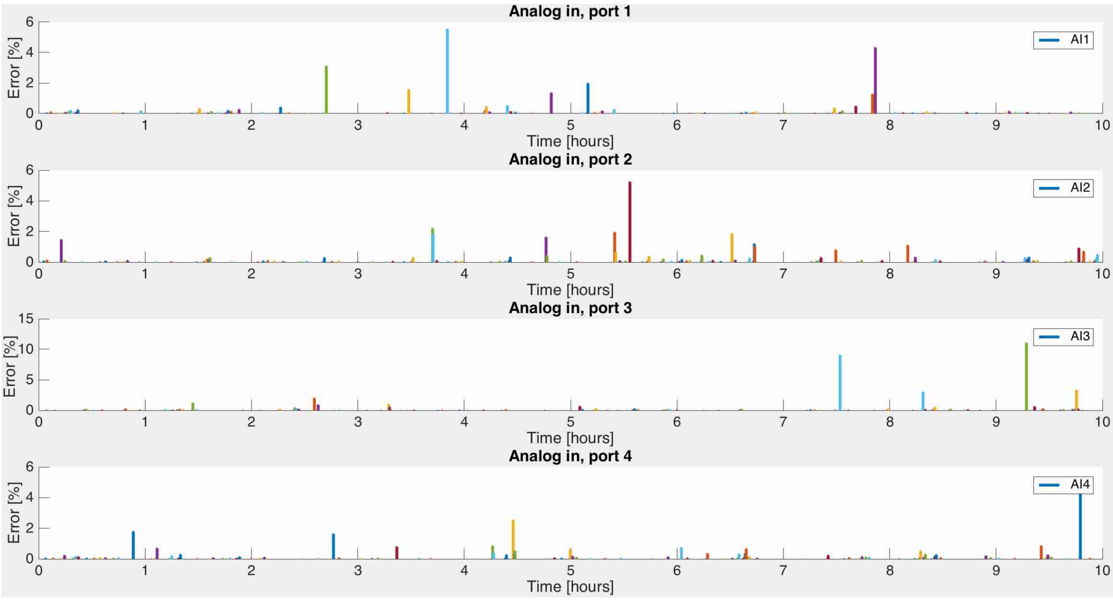


Figure 6.8: Graphical presentation in Matlab of the DI checksum errors and how they appear over time. The y-axis shows the percentage of errors since last error was detected, e.g. if two errors occur in a row it results in error percentage of 100%.

The errors from all four DI ports, collected for over 10 hours, can be seen graphed on Figure 6.9. The same conclusion can be derived from the error grouping on the digital in ports, that it does not seem to show grouping of errors and implies that the error is randomly distributed.

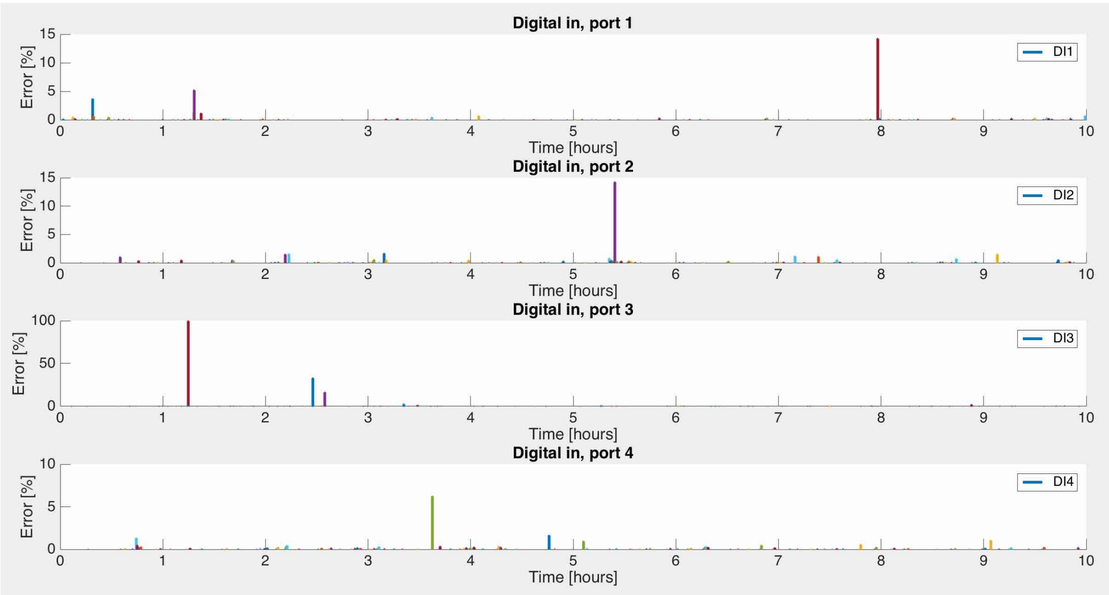


Figure 6.9: Graphical presentation in Matlab of the DI and AI checksum errors and how they appear over time. The y-axis shows the percentage of errors since last error was detected, e.g. if two errors occur in a row it results in error percentage of 100%.

Discussion

With the errors averaging of 0.054% means that there is 1 error per 1.852 readings. When error is detected the module responsible will register the last valid value instead. The errors do not seem to group at all ports at certain times on the modules. It therefore seems that the errors are randomly distributed over all ports.

When the Arduino modules were connected via USB to PC while logging then the errors were averaging about 0.5% instead of 0.054%. That means that about 10 times more errors occur with the USB connected than disconnected. The reason for that behavior is unclear but it is likely connected to the extra power it gets from the USB that creates interfering in the I^2C bus.

6.3.2 Digital out and analog out

The digital out and analog out commands come from the OPC-UA server to the Arduino. That means that the Arduino module is doing the checksum comparison to find errors in the bit string. To log the detected errors a terminal program was used for communication with the Arduino via serial. When an error occurred a string with the error percentage was sent over the serial to the terminal program and logged down. See Figure 6.10 an overview of the measurement setup.

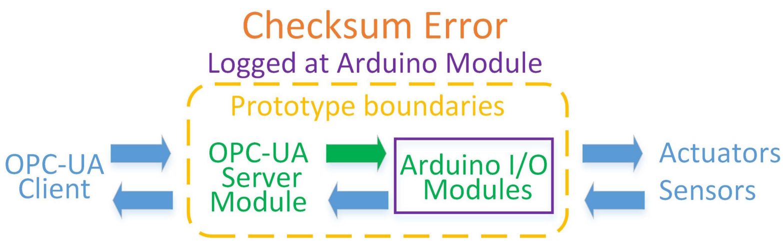


Figure 6.10: Overview of the measurement setup for the checksum error from OPC-UA server to Arduino actuator modules

It came clear early in the measurement process, when compering to DI and AI, that the error was higher than expected. It was similar to a problem encountered previously when logging the AI and DI errors while the Arduino modules where connected via USB. An extra measurement Arduino was used for measuring errors to counter the conflicting measurement way with the USB serial connection (see Section 4.2.3 for the measurement Arduino details). The program on the AO and DO modules was edited to output a signal on port 1 if an error was detected. The extra measurement Arduino was then used to measure and log the time when a signal was outputted from the AO and DO modules. The difference between error percentage with both measurement techniques, taken over period of 30 minutes, can be seen on Table 6.5.

Table 6.5: Total DO and AO checksum errors collected both through serial and through another measurement Arduino.

I/O	**Port**	**Error With Serial [%]**	**Error Without Serial [%]**
Analog out	1-2	0.419	0.139
Digital out	1-4	0.382	0.138
	Average	0.401	0.139

Discussion

It was possible to switch measurement technique by using additional Arduino for measuring the pin state change, timing and logging down. That way the serial monitoring could be avoided that produced about 2.9 times additional errors in the system. The errors on the DO and AO were more frequent than with AI and DI. The errors did average 0.139% for DO and AO and are therefore more than twice as frequent as the AI and DI.

6.4 Hardware reliability

6.4.1 Long term run

The prototype was more or less running for the last 2 months of the project for various tests and debugging purposes. It's longest, non-interrupted run, was 13 days with the OPC-UA client constantly writing and reading data with full load on the OPC-UA server. It did not crash after those 13 days but it had to be manually shutdown since it was to be used for other experiments. The prototype did, in fact, not crash or restart itself in any tests during the project. It is therefore safe to say that the prototype does seem reliable and meets long time run requirement of 10 days in a row without restarting. Longer test runs are however needed in future work to test its reliability further on.

6.4.2 Prototype power cut-off test

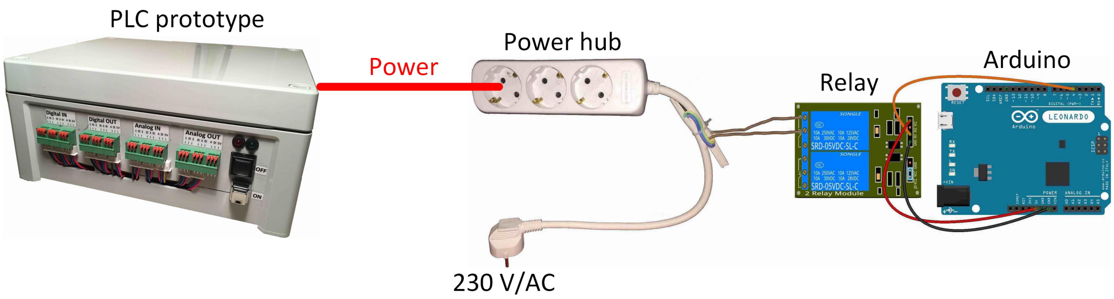


Figure 6.11: Overview of the experiment setup for the power cut-off test. Arduino controls a relay that is connected to a 230V power hub. The prototype is then connected to the power hub. The Arduino cuts the power for 1 minute every four minutes.

This experiment was done to test both the reliability of the SD memory card and other hardware components in the prototype. SD cards can end up corrupted when writing to it at the same moment when sudden power off occurs. There have numerous post been written on the Raspberry Pi Foundation forum about corruption of SD cards, it was therefore interesting to see whether it would affect the prototype. The Raspberry Pi Foundation does have instructions of how to avoid SD corruption where they recommend not overclocking the hardware, buy a genuine SD card, make sure that the power to the Raspberry Pi does not fall under 4.75 V and finally use a high quality USB cable [58]. All these point from the Raspberry Pi Foundation were applied to the prototype. The experiment setup can be seen on Figure 6.11 where an Arduino is used to control a rely that is connected to a power hub. The Arduino is programmed to cut the power completely and instantly to the prototype for 1 minute, it then turns the power on for 4 minutes. The LabVIEW OPC-UA client was then used to count how often it lost the connection to the prototype. See graph on Figure 6.12 with the prototype run time and average restart time over a 20 minute period. The prototype took about 47 seconds on average to restart it self including booting the OPC-UA server up. The maximum time it took to restart and boot up the OPC-UA server was about 54 s while the minimum time was about 35 s. That shows that the restart time does vary and is not predictable. After 355 successful restarts after a power cut-off, over the time span of 29.5 hours, the experiment was stopped with the SD card and all hardware intact. The experiment resulted in no SD card corruption or hardware damage and shows the prototype resilience to power cut-off situations. See the Arduino Code A.9 for the power cut-off experiment in Appendix.

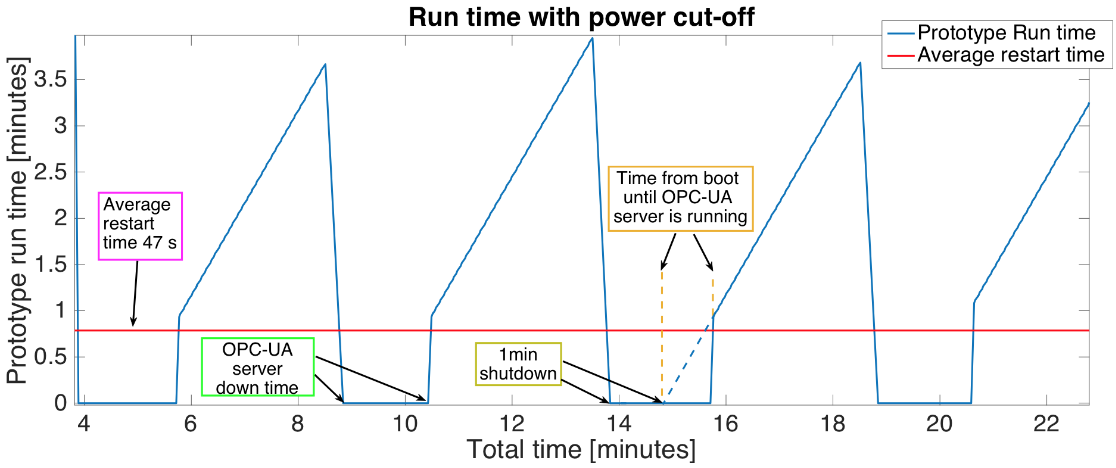


Figure 6.12: Graph of the prototype run time and the average restart time. It was logged down in the LabVIEW OPC-UA client while the power cut-off test was running.

6.5 Other results

The choice of hardware modules ended with Raspberry Pi 2 SoC for OPC-UA server module and Arduino Leonardo microcontrollers for I/O modules. Arduino is open source hardware and software by strictest definitions, with all schematics and software publicly available. Raspberry Pi 2 is, however, not an open source hardware by strictest definitions but was still chosen for its community support and favorable price. It runs on Linux which is an open source operating system. That means, for future work, that the open source OPC-UA server software that was developed on Raspberry Pi 2 can be ported to other SoC platforms running on Linux. The prototype was successfully built up to a beta stage where it is ready for further test in the hands of a person not involved in the development of the prototype.

The prototype was programmed with services that keep the OPC-UA server running after restart. Hardware shutdown and start buttons were successfully implemented. The requirement for modular design was met with the Arduino I/O modules connected to the Raspberry Pi OPC-UA server module via I^2C. The OPC-UA module supports up to 128 I/O modules on the I^2C bus. Four standard I/O modules were designed which included digital in, digital out, analog in and analog out. All I/O modules have 4 ports except the analog out module that has 2 ports. The reason being that it only had PMW instead of a real ADC and that required two additional DAC devices in addition to the analog out module. Only two DAC's were available to the author at the time of the thesis but adding two more should not be a problem.

A global communication protocol was successfully implemented for all communications between OPC-UA server module and all four I/O modules. A checksum error detection was integrated into the communication protocol which was used to filter out errors in the information string on the I^2C bus.

An open source OPC-UA server was successfully developed on Linux, running on Raspberry Pi 2. The OPC-UA server contained 17 variables that could be accessed via OPC-UA clients. LabVIEW OPC-UA client and UaExpert OPC-UA clients were used successfully for testing the prototype.

6.6 Additional results

The Raspberry Pi memory and CPU was monitored while the prototype was running at full load for 18 hours. It was interesting to see how the hardware was handling the prototype at full capacity. The amount of processing power the Raspberry Pi needs to handle under full load is observed in the graph on Figure 6.13. It shows that the average CPU usage is 57.61%. The memory usage was on average at 60%. That is the usage on full load and therefore seems that Raspberry Pi 2 hardware is fully capable of running the OPC-UA server.

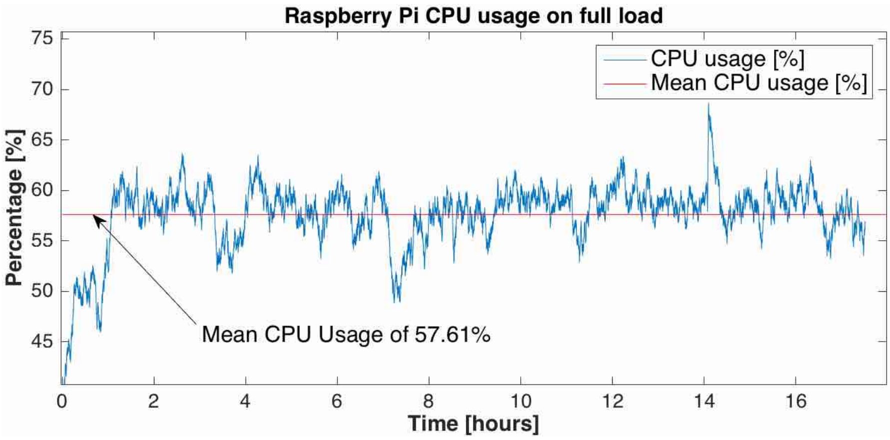


Figure 6.13: Graph of 311.863 sample points of CPU usage over the time-span of 18 hours. This was logged in the LabVIEW OPC-UA client in a full load on the system where the data was read/written as fast as possible.

When plotting all the analog in readings together for over 18 hours it was observed that at two occasions the readings looked to have leaked over (see graph on Figure 6.14). That could be because of hardware error in the Arduino in module or it could be that it is a value that passes through the checksum error detection. The readings also look like they fall over time on all four ports. These things need to be investigated further. Note that there was no leakage observed from the digital in module.

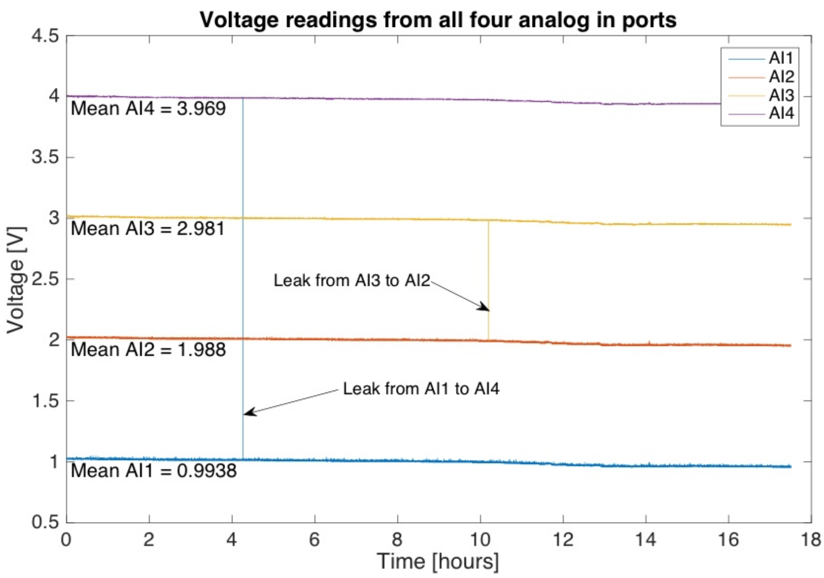


Figure 6.14: Graph of 311.863 sample points of all four analog inputs over the time-span of 18 hours. This was logged in the LabVIEW OPC-UA client in a full load on the system where the data was read/written as fast as possible.

6.7 Future work

6.7.1 Necessary improvements on the current prototype

The prototype is functional as it is but lacks proper input and output safety circuits to be robust enough for the industry. The safety circuit should protect the input and output pins from sudden surge of voltage and/or current.

6.7.2 Future extensions to the system

The current I/O modules only support AI, AO, DI and DO which are limited to 0-5 V. It is therefore recommended to develop more I/O modules that can handle different equipment that needs, for example, 4-20 mA or 0-10 V.

Because the Raspberry Pi 2 is only part open source it is recommended to scan the market for better suited SoC that is open source in the strictest sense. The minimum requirements are that it supports Linux and I^2C bus. The migration of the OPC-UA server to another Linux device should be relatively simple.

Chapter 7

Conclusion

The revolution of IoT, IIoT and Industry 4.0 is approaching with support from vendors like Siemens and Rockwell Automation, governments like Germany for Industry 4.0 and economic-political unions like the European Union [3],[18] . Raspberry Pi Foundation, Arduino, Sparkfun Electronics and Adafruit Industries host large open source communities that are involved with the current IoT/IIoT technologies and offer support to creators both in hardware and software.

IoT/IIOT and Industry 4.0 lacks support on the OSI application layer, it is therefore not vendor independent unless it has a unified communication protocol with it. The open source OPC-UA communication protocol was therefore chosen for its inbuilt security measures and support on the application layer. Raspberry Pi 2 was chosen as hardware for the OPC-UA server module and Arduino Leonardo for I/O modules. Raspberry Pi is not open source hardware by strictest definitions while Arduino has fully open source hardware and software. There were four I/O modules developed: Digital in, digital out, analog in and analog out. There were two additions needed for lack of components in the hardware. Real Time Clock (RTC) for the Raspberry Pi 2 and DAC device for Arduino analog out module.

The OPC-UA server on Raspberry Pi 2 was programmed with the open source OPC-UA software NodeOPCUA. It is based on the programming language Node.js and programmed on the Linux based Raspian Operating System [13]. All I/O modules were programmed in the open source Arduino IDE. An OPC-UA client was programmed in LabVIEW for testing the prototype. A free version of the UaExpert OPC-UA client program from Unified Automation was also used for certain tests [15].

A modular and OPC-UA based prototype was built with all functionalities defined in requirements with one exception where the analog out module has only two ports instead of four. All software running on the prototype is open source. The communication speed and reliability was measured. A test plan document with listed procedures was created for testing the prototype. A protocol with checksum error detection was developed for communication reliability between the OPC-UA server module and I/O modules. The prototype proved to have reliable communication and has been stable in longtime runs with no software/hardware crash on record. The communication speed from sensor to client (read) and client to actuator (write) was measured in LabVIEW with an average of less than 10 ms. That shows that the prototype can be used in projects that require fast response time.

The prototype did prove the concepts of building an open source hardware and software alternative to a commercial PLC. The prototype is currently limited to four I/O modules that operate from 0-5 V. Adapting the prototype by building circuits and functionality for I/O protections and different industrial standards would be a logical step in future iterations.

Appendix A

Appendix

1. Appendix A.1: Master’s Thesis task description
2. Appendix A.2: Master’s Thesis Abstract
3. Appendix A.3: Code
4. Appendix A.4: Test Plan Document

Telemark University College

Faculty of Technology

FMH606 Master's Thesis

Title: Open source hardware and software alternative to industrial PLC.

Version 1.0: 1-FEB-16

TUC supervisor: Nils-Olav Skeie

External partner: EFLA

EFLA is a general engineering and consulting company based in Iceland with international activities and consultancy around the globe with over 300 employees.
EFLA would like to work with HIT by providing and funding a project for master thesis.

Task background:
The industrial world has been using proven PLC's with specialized software for many years. In most cases that means paying monthly for software and being reliant on the PLC's manufacturer.

During the last years, micro-controllers have been more and more used by hobbyists in small integrated projects. If micro-controllers could be used in smaller, non-safety driven industrial projects with open source software and hardware, it would provide opportunities for engineering companies in making cheaper applications for smaller industrial projects.

This project is about researching, building and testing a prototype of a product that fits this description.

Task description:

1. Literature survey of the industrial PLC landscape with emphasis on why open source hardware/software implementation could be beneficial.
2. Literature survey of the OPC-UA communication standard with emphasis on security and include comparison with the older OPC standard.
3. Select a set of hardware components that can be used for building a prototype and state the reason for each chosen components
4. Focus on a solution that is modular and easy to implement
5. Build a prototype using the selected hardware components and OPC UA as communication protocol
6. Make a test plan
7. Test the prototype with focus on reliability and communication speed

Student category:

SCE student having work experience with EFLA.

Practical arrangements:

EFLA will get hold of the necessary hardware and software for the project.

Signatures:

Student (date and signature): Sturla Rúnarsson

Supervisor (date and signature): [illegible] 2 feb 16

University College of Southeast Norway

MASTER'S THESIS, COURSE CODE FMH606

Student:	**Sturla Rúnarsson**
Thesis title:	**Open Source Hardware and Software Alternative to Industrial PLC**
Signature:	. . . .
Number of pages:	130
Keywords:	**Prototyping, OPC-UA, Arduino, Raspberry Pi,** **I2C, PLC, Node.js, Open Source.**

Supervisor:	Nils-Olav Skeie	Sign.:	. . . .
2nd supervisor:	<name>	Sign.:	. . . .
Censor:	Håkon Tjelland	Sign.:	. . . .
External partner:	EFLA	Sign.:	. . . .
Availability:	Open		

Archive approval (supervisor signature): Sign.: . . . . **Date :** . . . .

Abstract:

The revolution of Internet of Things (IoT), Industrial Internet of Things (IIoT) and Industry 4.0 is approaching with new market opportunities for all kinds of smart devices. This thesis was about building such a device, an open source hardware and software alternative to industrial Programmable Logic Controller (PLC). The idea was to prove the concepts by building a prototype with a solid foundation that includes the best suiting communication protocol available (OPC-UA) and a modular functionality for ease of repair and customisation. The challenges were appropriate choices of open source hardware and software along with making sure that interactions between different parts of hardware were possible. That includes the communication protocols between modules and extensive programming needed in various programming languages. Several tests were then needed to validate the required communication speed and reliability requirements.

The prototype was developed with Open Platform Communications - Unified Architecture (OPC- UA) server module and four input/output (I/O) modules which include digital in, digital out, analog in and analog out. Raspberry Pi 2 was chosen as the System on Chip (SoC) hardware capable of running on Linux and hosting the OPC-UA server while Arduino Leonardo microcontrollers were chosen for the I/O modules. The OPC-UA server on the SoC hardware was programmed in node.js on Linux while all I/O microcontrollers were programmed in a subset of C/C++. OPC-UA client in LabVIEW was developed for the majority of experiments while Matlab was used for data analysis.

The concept of building an open source PLC prototype was proven and its capabilities tested. The prototype proved to be stable in long time runs with no software/hardware crash on record. The communication speed from sensor to client (read) and client to actuator (write) was measured in LabVIEW with an average of under 10 ms. Only open source hardware and software was used, except for Raspberry Pi SoC OPC-UA server module which is not defined as an open source hardware by strictest definition. Modular I/O functionality was successfully implemented on a I2C communication bus. The prototype shows potential for practical use and is ready for further development with emphasis on pin protections and upgrading I/O modules to industrialized standards for sensors and actuators.

A.3 Code

A.3.1 OPC-UA server Node.js code

Code A.1: OPC-UA server code that is written in Node.js and runs on Linux in Raspberry Pi hardware.

```
/*
 * Created by Sturla on 20/01/16.
 * Last updated 25/04/16.
 */


var fs = require('fs');
//Require the node-opcua library
var opcua = require("node-opcua");
//Require the os library for accessing the Raspberry Pi OS information
var os = require("os");

//Require I2C library
var i2c = require('i2c');
var address = 0x18;

// Require sys and child_process for restarting with a button
var sys = require('sys')
var exec = require('child_process').exec;

// Require GPIO library for the buttons and LEDs
var Gpio = require('onoff').Gpio;

ledON = new Gpio(5,'out'); //Green LED
ledOFF = new Gpio(6,'out'); //Red LED

buttonShutdown = new Gpio(19, 'in', 'both'); //Initializing the input on port 19

// Set counter so that the button wont recognize pressing
// until the button is pressed for a moment
var counterShutdown=0;

// For shutdown the Rpi with a button
buttonShutdown.watch(function(err, value) {
    if (counterShutdown>2)
    {
        ledON.writeSync(0); //Set green LED off
        ledOFF.writeSync(1); //Set red light on
        function puts(error, stdout, stderr) { sys.puts(stdout) }
        exec("sudo shutdown -h now", puts);
        counterShutdown=0;
    }
    ++counterShutdown;
});

ledON.writeSync(1); //Set green LED on
ledOFF.writeSync(0); //Set red LED off

//Set addresses for all the i2c devices used
var Arduino_Analog_IN = new i2c(address, {device: '/dev/i2c-1'});
Arduino_Analog_IN.setAddress(0x5);

var Arduino_Digital_IN = new i2c(address, {device: '/dev/i2c-1'});
Arduino_Digital_IN.setAddress(0x6);

```

```
var Arduino_Digital_OUT = new i2c(address, {device: '/dev/i2c-1'});
Arduino_Digital_OUT.setAddress(0x7);

var Arduino_Analog_OUT = new i2c(address, {device: '/dev/i2c-1'});
Arduino_Analog_OUT.setAddress(0x8);

//Set variables used
var Digital_IN_1=0;
var Digital_IN_2=0;
var Digital_IN_3=0;
var Digital_IN_4=0;

var Digital_OUT_1="0";
var Digital_OUT_2="0";
var Digital_OUT_3="0";
var Digital_OUT_4="0";

var Analog_IN_1=0;
var Analog_IN_2=0;
var Analog_IN_3=0;
var Analog_IN_4=0;

var Analog_OUT_1="0";
var Analog_OUT_2="0";
var AO1protoVector=[14];
var AO2protoVector=[14];
var AO1Checksum;
var AO2Checksum;

var DO1protoVector=[14];
var DO2protoVector=[14];
var DO3protoVector=[14];
var DO4protoVector=[14];
var DO1Checksum;
var DO2Checksum;
var DO3Checksum;
var DO4Checksum;

var counterDI1=0;
var counterDI2=0;
var counterDI3=0;
var counterDI4=0;

var counterAI1=0;
var counterAI2=0;
var counterAI3=0;
var counterAI4=0;

var scan;

/*
 To set username and password, need to change in "/home/pi/MyOPCUAserver/node_modules/
     node-opcua/lib/server/opcua_server.js":
 options.allowAnonymous = ( options.allowAnonymous === undefined) ? true : options.
     allowAnonymous;
 to
 options.allowAnonymous = false; //( options.allowAnonymous === undefined) ? true :
     options.allowAnonymous;
 Then the anonymouse login is not longer available
 */
var userManager = {
    isValidUser: function (userName, password) {

```

```
        if (userName === "user1" && password === "password1") {
            return true;
        }
        if (userName === "user2" && password === "password2") {
            return true;
        }
        return false;
    }
}

//Set the path for the certificates
var path = require("path");
var server_certificate_file            = path.join("/home/pi/node_modules/node-opcua/
    certificates/server_selfsigned_cert_1024.pem");
var server_certificate_privatekey_file = path.join("/home/pi/node_modules/node-opcua/
    certificates/server_key_1024.pem");

var get_fully_qualified_domain_name = opcua.get_fully_qualified_domain_name;
var makeApplicationUrn = opcua.makeApplicationUrn;

// Create an instance of OPCUAServer
var server = new opcua.OPCUAServer({
    port: 4840, // the port of the listening socket of the server
    //resourcePath: "UA/MyLittleServer", // this path will be added to the endpoint
        resource name
    buildInfo : {
        productName: "NodeOPCUA",
        buildNumber: "1006",
        buildDate: new Date(2016,3,10)
    },
    certificateFile: server_certificate_file,
    privateKeyFile: server_certificate_privatekey_file,

    serverInfo: {
        applicationUri: makeApplicationUrn(get_fully_qualified_domain_name(), "
            NodeOPCUA-Server"),
        productUri: "NodeOPCUA-Server",
        applicationName: {text: "NodeOPCUA" ,locale:"en"},
        gatewayServerUri: null,
        discoveryProfileUri: null,
        discoveryUrls: []
    },

    allowAnonymouse: false, // Seems to have no affect
    userManager: userManager, // Set userManager variable to the server settings
    isAuditing: true
});



function post_initialize() {
    console.log("initialized");

    function construct_my_address_space(server) {

        var addressSpace = server.engine.addressSpace;

        // Declare new objects, basically just folders for variables to be separated
            by
        //Declare a new object, info from the Raspberry Pi 2
        var Rpi_info = addressSpace.addObject({
            organizedBy: addressSpace.rootFolder.objects,
            browseName: "Rpi_info"
```

```
    });
    // declare a new object, Digital in
    var Digital_in = addressSpace.addObject({
        organizedBy: addressSpace.rootFolder.objects,
        browseName: "Digital_in"
    });
    // declare a new object, Digital out
    var Digital_out = addressSpace.addObject({
        organizedBy: addressSpace.rootFolder.objects,
        browseName: "Digital_out"
    });
    // declare a new object, Analog in
    var Analog_in = addressSpace.addObject({
        organizedBy: addressSpace.rootFolder.objects,
        browseName: "Analog_in"
    });
    // declare a new object, Analog out
    var Analog_out = addressSpace.addObject({
        organizedBy: addressSpace.rootFolder.objects,
        browseName: "Analog_out"
    });

    //------------Collections of Checksum Calculation Functions---------------BEGIN

    //Scan the I2C addresses (good to observe if there are problems with the server)
    scan = Arduino_Analog_IN.scan(function(err, data) {
        // result contains an array of addresses
    });

    //Sums up the values in an array, from ASCII (number plus 48)
    function sumArray(array) {
        for (
            var
                index = 0,               // The iterator
                length = array.length,   // Cache the array length
                sum = 0;                 // The total amount
            index < length;          // The "for"-loop condition

            sum += getNum(array[index++]) // Add number on each iteration
        );

        return sum;
    }

    //Checks if the value is a number and returns 0 in ASCII if NaN (48+number is ASCII value)
    function getNum(val) {
        if (isNaN(val)) {
            return 48;
        }
        return val+48;
    }

    //Function for sum up array strings
    function sumArrayStr(array) {
        for (
            var
                index = 0,               //The iterator
                length = array.length,   //Cache the array length
                sum = 0;                 //The total amount
            index < length;          //The "for"-loop condition
```

```

                sum += getNum(parseInt(array[index++])) //Add number on each iteration
            );

            return sum;
        }

        //Function for extracting data from the buffer from Digital IN
        function CheckSumDI(buffer) {
            var JSONstring = JSON.stringify(buffer); //Create JSON string from buffer
            //Extract the data part from the string
            var n1 = JSONstring.indexOf("[");
            var n2 = JSONstring.lastIndexOf("]");
            var ByteString = JSONstring.substr(n1+1,(n2-n1)-1);
            //Extract all the messagebytes and the Checksum from Arduino
            var Messagebyte1 = ByteString.split(",")[4];
            var Messagebyte2  = ByteString.split(",")[5];
            var Messagebyte3  = ByteString.split(",")[6];
            var Messagebyte4  = ByteString.split(",")[7];
            var Messagebyte5  = ByteString.split(",")[8];
            var Messagebyte6  = ByteString.split(",")[9];
            var Messagebyte7  = ByteString.split(",")[10];
            var Messagebyte8  = ByteString.split(",")[11];
            var MessageLength  = ByteString.split(",")[12];
            var ChecksumSlave = ByteString.split(",")[13];
            //Sum all parts in the array except the checksum at the end of the bytestring
            var DI1allArray = ByteString.split(",");
            DI1allArray.splice(13,1);
            for (
                var
                    index = 0,              // The iterator
                    length = DI1allArray.length,  // Cache the array length
                    sum = 0;                // The total amount
                index < length;         // The "for"-loop condition

                sum += getNum(parseInt(DI1allArray[index++])) // Add number on each iteration
            );
            var ChecksumRemainder = sum % 64; //Take the remainder of the sum
            return [Messagebyte1,Messagebyte2,Messagebyte3,Messagebyte4,Messagebyte5,Messagebyte6,Messagebyte7,Messagebyte8,MessageLength,ChecksumRemainder,ChecksumSlave,];
        }

        //Compares the checksum from slave to the calculated checksum, returns 1 for true and 0 for false
        function ChecksumComparison(Cs1,Cs2) {
            if (Cs1 == Cs2)
            {
                var TrueFalse = 1;
            }
            else if (Cs1 != Cs2)
            {
                var TrueFalse = 0;
            }
            return TrueFalse;
        }

        //Get the right length of value from the array, using the length byte
        function GetValue(DataVector) {
            var Value;
            if (DataVector[8]==1)
```

```
            {
                //var Value = DataVector[0];
                Value = DataVector[0];
            }
            else if (DataVector[8]==2)
            {
                //var Value = DataVector[0].concat(DataVector[1]);
                Value = DataVector[0]+DataVector[1];
            }
            else if (DataVector[8]==3)
            {
                //var Value = DataVector[0].concat(DataVector[1],DataVector[2]);
                Value = DataVector[0]+DataVector[1]+DataVector[2];
            }
            else if (DataVector[8]==4)
            {
                //var Value = DataVector[0].concat(DataVector[1],DataVector[2],
                    DataVector[3]);
                Value = DataVector[0]+DataVector[1]+DataVector[2]+DataVector[3];
            }
            return Value;
        }
//------------Collections of Checksum Calculation Functions---------------END

//------------Add variables---------------Begin
        //Note that the text that is commented out in this part is relevant to the
            measurements and logging
        //of the checksum errors.

        // Add variables to the address space
        addressSpace.addVariable({
            componentOf: Digital_in,
            browseName: "Digital_IN_1",
            dataType: "Float",
            value: {
                get: function () {
                    Arduino_Digital_IN.readBytes(1,14, function(err,DI1) {
                        var DataVectorDI1 = CheckSumDI(DI1); //Get data from buffer
                        var IsPacketloss = ChecksumComparison(DataVectorDI1[9],
                            DataVectorDI1[10]); //Is packetloss
                        //console.log(DI1);
                        //console.log(DataVectorDI1);
                        //console.log(IsPacketloss);
                        if (IsPacketloss == 1) //If no packet loss then
                        {
                            Digital_IN_1 = DataVectorDI1[0];
                            //console.log("CheckSum Passed!");
                            //console.log(Digital_IN_1);
                            counterDI1 +=1;
                        }
                        /*
                         if (IsPacketloss == 0) //If packet loss then
                         {
                         //console.log("Packet loss!");
                         console.log("DI1: ",1/counterDI1);

                         fs.appendFile('DigitalIN1.txt',[1/counterDI1 + '\n']  ,
                             function (err) {
                         if (err) throw err;
                         //console.log('It\'s saved! in same location.');
                         });
                         counterDI1=0;
                         }
```

```
                    */
                });
                return new opcua.Variant({dataType: opcua.DataType.Float, value: Digital_IN_1 });
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Digital_in,
        browseName: "Digital_IN_2",
        dataType: "Float",
        value: {
            get: function () {
                Arduino_Digital_IN.readBytes(2,14, function(err,DI2) {
                    var DataVectorDI2 = CheckSumDI(DI2); //Get data from buffer
                    var IsPacketloss = ChecksumComparison(DataVectorDI2[9], DataVectorDI2[10]); //Is packetloss
                    //console.log(DataVectorDI2);
                    //console.log(IsPacketloss);
                    //console.log(DI2);
                    if (IsPacketloss == 1) //If no packet loss then
                    {
                        Digital_IN_2 = DataVectorDI2[0];
                        //console.log("CheckSum Passed!");
                        //console.log(Digital_IN_2);
                        counterDI2 +=1;
                    }
                    /*
                     if (IsPacketloss == 0) //If packet loss then
                     {
                     //console.log("Packet loss!");
                     console.log("DI2: ",1/counterDI2);

                     fs.appendFile('DigitalIN2.txt',[1/counterDI2 + '\n']  , function (err) {
                     if (err) throw err;
                     //console.log('It\'s saved! in same location.');
                     });
                     counterDI2=0;
                     }
                     */
                });
                return new opcua.Variant({dataType: opcua.DataType.Float, value: Digital_IN_2 });
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Digital_in,
        browseName: "Digital_IN_3",
        dataType: "Float",
        value: {
            get: function () {
                Arduino_Digital_IN.readBytes(3,14, function(err,DI3) {
                    var DataVectorDI3 = CheckSumDI(DI3); //Get data from buffer
                    var IsPacketloss = ChecksumComparison(DataVectorDI3[9], DataVectorDI3[10]); //Is packetloss
                    //console.log(DataVectorDI3);
                    //console.log(IsPacketloss);
                    //console.log(DI3);
                    if (IsPacketloss == 1) //If no packet loss then
```

```
                        {
                            Digital_IN_3 = DataVectorDI3[0];
                            //console.log("CheckSum Passed!");
                            //console.log(Digital_IN_3);
                            counterDI3 +=1;
                        }
                        /*
                         if (IsPacketloss == 0) //If packet loss then
                         {
                         //console.log("Packet loss!");
                         console.log("DI3: ",1/counterDI3);

                         fs.appendFile('DigitalIN3.txt',[1/counterDI3 + '\n']  , function (err) {
                         if (err) throw err;
                         //console.log('It\'s saved! in same location.');
                         });
                         counterDI3=0;
                         }
                         */

                    });
                    return new opcua.Variant({dataType: opcua.DataType.Float, value: Digital_IN_3});
                }
            }
        });

        addressSpace.addVariable({
            componentOf: Digital_in,
            browseName: "Digital_IN_4",
            dataType: "Float",
            value: {
                get: function () {
                    Arduino_Digital_IN.readBytes(4,14, function(err,DI4) {
                        var DataVectorDI4 = CheckSumDI(DI4); //Get data from buffer
                        var IsPacketloss = ChecksumComparison(DataVectorDI4[9], DataVectorDI4[10]); //Is packetloss
                        //console.log(DataVectorDI4);
                        //console.log(IsPacketloss);
                        //console.log(DI4);
                        if (IsPacketloss == 1) //If no packet loss then
                        {
                            Digital_IN_4 = DataVectorDI4[0];
                            //console.log("CheckSum Passed!");
                            //console.log(Digital_IN_4);
                            counterDI4 +=1;
                        }
                        /*
                         if (IsPacketloss == 0) //If packet loss then
                         {
                         //console.log("Packet loss!");
                         console.log("DI4: ",1/counterDI4);

                         fs.appendFile('DigitalIN4.txt',[1/counterDI4 + '\n']  , function (err) {
                         if (err) throw err;
                         //console.log('It\'s saved! in same location.');
                         });
                         counterDI4=0;
                         }
                         */
                    });
```

```
                    return new opcua.Variant({dataType: opcua.DataType.Float, value: Digital_IN_4 });
                }
            }
        });

        addressSpace.addVariable({
            componentOf: Digital_out,
            nodeId: "ns=1;b=1020DA",
            browseName: "Digital_OUT_1",
            dataType: "String",
            value: {
                get: function () {
                    return new opcua.Variant({dataType: opcua.DataType.String, value: Digital_OUT_1 });
                },
                set: function (variant) {

                    //Set the protocol vector
                    DO1protoVector=[1,2,1,0,parseInt(variant.value.substring(0,1)),parseInt(variant.value.substring(1,2)),parseInt(variant.value.substring(2,3)),parseInt(variant.value.substring(3,4)),parseInt(variant.value.substring(4,5)),parseInt(variant.value.substring(5,6)),parseInt(variant.value.substring(6,7)),parseInt(variant.value.substring(7,8)),variant.value.length];
                    //Sum up the protocol vector (Checksum)
                    DO1Checksum = sumArray(DO1protoVector);
                    //Add the remainder to the end of the protocol vector
                    DO1protoVector.push(DO1Checksum % 64);

                    //console.log("Push sum : ", DO1protoVector);
                    //console.log("Checksum remainder: ", DO1Checksum % 64);
                    //console.log("Checksum: ", DO1Checksum);
                    //Write to the Arduino
                    //Arduino_Digital_OUT.write(DO1protoVector, function(err) { console.log("New Digital out 1 value is: ",variant.value); });
                    Arduino_Digital_OUT.write(DO1protoVector, function(err) { });
                    Digital_OUT_1=variant.value;
                    return opcua.StatusCodes.Good;
                }
            }
        });

        addressSpace.addVariable({
            componentOf: Digital_out,
            nodeId: "ns=2;b=1020DB",
            browseName: "Digital_OUT_2",
            dataType: "String",
            value: {
                get: function () {
                    return new opcua.Variant({dataType: opcua.DataType.String, value: Digital_OUT_2 });
                },
                set: function (variant) {
                    //Set the protocol vector
                    DO2protoVector=[1,2,2,0,parseInt(variant.value.substring(0,1)),parseInt(variant.value.substring(1,2)),parseInt(variant.value.substring(2,3)),parseInt(variant.value.substring(3,4)),parseInt(variant.value.substring(4,5)),parseInt(variant.value.substring(5,6)),parseInt(variant.value.substring(6,7)),parseInt(variant.value.substring(7,8)),variant.value.length];
                    //Sum up the protocol vector (Checksum)
```

```
                    DO2Checksum = sumArray(DO2protoVector);
                    //Add the remainder to the end of the protocol vector
                    DO2protoVector.push(DO2Checksum % 64);

                    //console.log("Push sum : ", DO2protoVector);
                    //console.log("Checksum remainder: ", DO2Checksum % 64);
                    //console.log("Checksum: ", DO2Checksum);
                    //Write to the Arduino
                    //Arduino_Digital_OUT.write(DO2protoVector, function(err) { console.log("New Digital out 2 value is: ",variant.value); });
                    Arduino_Digital_OUT.write(DO2protoVector, function(err) { });
                    Digital_OUT_2=variant.value;
                    return opcua.StatusCodes.Good;
                }
            }
        });

        addressSpace.addVariable({
            componentOf: Digital_out,
            nodeId: "ns=3;b=1020DC",
            browseName: "Digital_OUT_3",
            dataType: "String",
            value: {
                get: function () {
                    return new opcua.Variant({dataType: opcua.DataType.String, value: Digital_OUT_3 });
                },
                set: function (variant) {
                    //Set the protocol vector
                    DO3protoVector=[1,2,3,0,parseInt(variant.value.substring(0,1)),parseInt(variant.value.substring(1,2)),parseInt(variant.value.substring(2,3)),parseInt(variant.value.substring(3,4)),parseInt(variant.value.substring(4,5)),parseInt(variant.value.substring(5,6)),parseInt(variant.value.substring(6,7)),parseInt(variant.value.substring(7,8)),variant.value.length];
                    //Sum up the protocol vector (Checksum)
                    DO3Checksum = sumArray(DO3protoVector);
                    //Add the remainder to the end of the protocol vector
                    DO3protoVector.push(DO3Checksum % 64);

                    //console.log("Push sum : ", DO3protoVector);
                    //console.log("Checksum remainder: ", DO3Checksum % 64);
                    //console.log("Checksum: ", DO3Checksum);
                    //Write to the Arduino
                    //Arduino_Digital_OUT.write(DO3protoVector, function(err) { console.log("New Digital out 3 value is: ",variant.value); });
                    Arduino_Digital_OUT.write(DO3protoVector, function(err) { });
                    Digital_OUT_3=variant.value;
                    return opcua.StatusCodes.Good;
                }
            }
        });

        addressSpace.addVariable({

            componentOf: Digital_out,
            nodeId: "ns=4;b=1020DD",
            browseName: "Digital_OUT_4",
            dataType: "String",
            value: {
                get: function () {
```

```
                return new opcua.Variant({dataType: opcua.DataType.String, value: Digital_OUT_4 });
            },
            set: function (variant) {
                //Set the protocol vector
                DO4protoVector=[1,2,4,0,parseInt(variant.value.substring(0,1)),parseInt(variant.value.substring(1,2)),parseInt(variant.value.substring(2,3)),parseInt(variant.value.substring(3,4)),parseInt(variant.value.substring(4,5)),parseInt(variant.value.substring(5,6)),parseInt(variant.value.substring(6,7)),parseInt(variant.value.substring(7,8)),variant.value.length];
                //Sum up the protocol vector (Checksum)
                DO4Checksum = sumArray(DO4protoVector);
                //Add the remainder to the end of the protocol vector
                DO4protoVector.push(DO4Checksum % 64);

                //console.log("Push sum : ", DO4protoVector);
                //console.log("Checksum remainder: ", DO4Checksum % 64);
                //console.log("Checksum: ", DO4Checksum);
                //Write to the Arduino
                //Arduino_Digital_OUT.write(DO4protoVector, function(err) { console.log("New Digital out 4 value is: ",variant.value); });
                Arduino_Digital_OUT.write(DO4protoVector, function(err) { });
                Digital_OUT_4=variant.value;
                return opcua.StatusCodes.Good;
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Analog_in,
        browseName: "Analog_IN_1",
        //dataType: "Double",
        dataType: "Float",
        value: {
            get: function () {
                Arduino_Analog_IN.readBytes(1,14, function(err,AI1) {

                    var DataVectorAI1 = CheckSumDI(AI1); //Get data from buffer
                    var IsPacketloss = ChecksumComparison(DataVectorAI1[9], DataVectorAI1[10]); //Is packetloss
                    //console.log("Datavector: ", DataVectorAI1);
                    //console.log("Packet loss: ", IsPacketloss);
                    //console.log(AI1);
                    //console.log(scan);
                    if (IsPacketloss == 1) //If no packet loss then
                    {
                        Analog_IN_1 = GetValue(DataVectorAI1);
                        //console.log("CheckSum Passed!");
                        //console.log(Analog_IN_1);
                        counterAI1 +=1;
                    }
                    /*
                     if (IsPacketloss == 0) //If packet loss then
                     {
                     //console.log("Packet loss!");
                     console.log("AI1: ",1/counterAI1);

                     fs.appendFile('AnalogIN1.txt',[1/counterAI1 + '\n']   , function (err) {
                     if (err) throw err;
                     //console.log('It\'s saved! in same location.');
```

```
                    });
                    counterAI1=0;
                    }
                    */
                });
                return new opcua.Variant({dataType: opcua.DataType.Float, value:
                    Analog_IN_1 });
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Analog_in,
        browseName: "Analog_IN_2",
        //dataType: "Double",
        dataType: "Float",
        value: {
            get: function () {
                Arduino_Analog_IN.readBytes(2,14, function(err,AI2) {
                    var DataVectorAI2 = CheckSumDI(AI2); //Get data from buffer
                    var IsPacketloss = ChecksumComparison(DataVectorAI2[9],
                        DataVectorAI2[10]); //Is packetloss
                    //console.log("Datavector: ", DataVectorAI2);
                    //console.log("Packet loss: ", IsPacketloss);
                    //console.log(AI2);
                    if (IsPacketloss == 1) //If no packet loss then
                    {
                        Analog_IN_2 = GetValue(DataVectorAI2);
                        //console.log("CheckSum Passed!");
                        //console.log(Analog_IN_2);
                        counterAI2 +=1;
                    }
                    /*
                    if (IsPacketloss == 0) //If packet loss then
                    {
                    //console.log("Packet loss!");
                    console.log("AI2: ",1/counterAI2);

                    fs.appendFile('AnalogIN2.txt',[1/counterAI2 + '\n']  ,
                        function (err) {
                    if (err) throw err;
                    //console.log('It\'s saved! in same location.');
                    });
                    counterAI2=0;
                    }
                    */
                });
                return new opcua.Variant({dataType: opcua.DataType.Float, value:
                    Analog_IN_2 });
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Analog_in,
        browseName: "Analog_IN_3",
        //dataType: "Double",
        dataType: "Float",
        value: {
            get: function () {
                Arduino_Analog_IN.readBytes(3,14, function(err,AI3) {
                    var DataVectorAI3 = CheckSumDI(AI3); //Get data from buffer
```

```
                var IsPacketloss = ChecksumComparison(DataVectorAI3[9], DataVectorAI3[10]); //Is packetloss
                //console.log("Datavector: ", DataVectorAI3);
                //console.log("Packet loss: ", IsPacketloss);
                //console.log(AI3);
                if (IsPacketloss == 1) //If no packet loss then
                {
                    Analog_IN_3 = GetValue(DataVectorAI3);
                    //console.log("CheckSum Passed!");
                    //console.log(Analog_IN_3);
                    counterAI3 +=1;
                }
                /*
                 if (IsPacketloss == 0) //If packet loss then
                 {
                 //console.log("Packet loss!");
                 console.log("AI3: ",1 / counterAI3);

                 fs.appendFile('AnalogIN3.txt', [1 / counterAI3 + '\n'], function (err) {
                 if (err) throw err;
                 //console.log('It\'s saved! in same location.');
                 });
                 counterAI3 = 0;
                 }
                 */
            });
            return new opcua.Variant({dataType: opcua.DataType.Float, value: Analog_IN_3 });
        }
    }
});

addressSpace.addVariable({
    componentOf: Analog_in,
    browseName: "Analog_IN_4",
    //dataType: "Double",
    dataType: "Float",
    value: {
        get: function () {
            Arduino_Analog_IN.readBytes(4,14, function(err,AI4) {
                var DataVectorAI4 = CheckSumDI(AI4); //Get data from buffer
                var IsPacketloss = ChecksumComparison(DataVectorAI4[9], DataVectorAI4[10]); //Is packetloss
                //console.log("Datavector: ", DataVectorAI4);
                //console.log("Packet loss: ", IsPacketloss);
                //console.log(AI4);
                if (IsPacketloss == 1) //If no packet loss then
                {
                    Analog_IN_4 = GetValue(DataVectorAI4);
                    //console.log("CheckSum Passed!");
                    //console.log(Analog_IN_4);
                    counterAI4 +=1;
                }
                /*
                 if (IsPacketloss == 0) //If packet loss then
                 {
                 //console.log("Packet loss!");
                 console.log("AI4: ",1 / counterAI4);

                 fs.appendFile('AnalogIN4.txt', [1 / counterAI4 + '\n'], function (err) {
                 if (err) throw err;
```

```
                    //console.log('It\'s saved! in same location.');
                    });
                    counterAI4 = 0;
                    }
                    */
                });
                return new opcua.Variant({dataType: opcua.DataType.Float, value:
                    Analog_IN_4 });
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Analog_out,
        nodeId: "ns=5;b=1020AA",
        browseName: "Analog_OUT_1",
        dataType: "String",
        value: {
            get: function () {
                return new opcua.Variant({dataType: opcua.DataType.String, value:
                    Analog_OUT_1 });
            },
            set: function (variant) {

                //Set the protocol vector
                AO1protoVector=[1,1,1,0,parseInt(variant.value.substring(0,1)),
                    parseInt(variant.value.substring(1,2)),parseInt(variant.value
                    .substring(2,3)),parseInt(variant.value.substring(3,4)),
                    parseInt(variant.value.substring(4,5)),parseInt(variant.value
                    .substring(5,6)),parseInt(variant.value.substring(6,7)),
                    parseInt(variant.value.substring(7,8)),variant.value.length];
                //Sum up the protocol vector (Checksum)
                AO1Checksum = sumArray(AO1protoVector);
                //Add the remainder to the end of the protocol vector
                AO1protoVector.push(AO1Checksum % 64);

                //console.log("Push sum : ", AO1protoVector);
                //console.log("Checksum remainder: ", AO1Checksum % 64);
                //console.log("Checksum3: ", AO1Checksum);
                //Write to the Arduino
                //Arduino_Analog_OUT.write(AO1protoVector, function(err) { console
                    .log("New Analog out 1 value is: ",variant.value); });
                Arduino_Analog_OUT.write(AO1protoVector, function(err) { });

                Analog_OUT_1=variant.value;

                return opcua.StatusCodes.Good;
            }
        }
    });

    addressSpace.addVariable({
        componentOf: Analog_out,
        nodeId: "ns=6;b=1020AB",
        browseName: "Analog_OUT_2",
        dataType: "String",
        value: {
            get: function () {
                return new opcua.Variant({dataType: opcua.DataType.String, value:
                    Analog_OUT_2 });
            },
            set: function (variant) {

```

```
                //Set the protocol vector
                AO2protoVector=[1,1,2,0,parseInt(variant.value.substring(0,1)),parseInt(variant.value.substring(1,2)),parseInt(variant.value.substring(2,3)),parseInt(variant.value.substring(3,4)),parseInt(variant.value.substring(4,5)),parseInt(variant.value.substring(5,6)),parseInt(variant.value.substring(6,7)),parseInt(variant.value.substring(7,8)),variant.value.length];
                //Sum up the protocol vector (Checksum)
                AO2Checksum = sumArray(AO2protoVector);
                //Add the remainder to the end of the protocol vector
                AO2protoVector.push(AO2Checksum % 64);

                //console.log("Push sum : ", AO2protoVector);
                //console.log("Checksum remainder: ", AO2Checksum % 64);
                //console.log("Checksum3: ", AO2Checksum);
                //Write to the Arduino
                //Arduino_Analog_OUT.write(AO2protoVector, function(err) { console.log("New Analog out 1 value is: ",variant.value); });
                Arduino_Analog_OUT.write(AO2protoVector, function(err) { });

                Analog_OUT_2=variant.value;
                return opcua.StatusCodes.Good;
            }
        }
    });

    // Percentage of Memory Used by Rpi
    server.nodeVariable1 = addressSpace.addVariable({
        componentOf: Rpi_info,
        nodeId: "ns=23;b=1020AD",
        browseName: "Percentage Memory Used",
        dataType: "Double",
        minimumSamplingInterval: 1000,
        value: {
            get: function () {
                var percentageMemUsed = 1.0 - (os.freemem() / os.totalmem() );
                var value = percentageMemUsed * 100;
                return new opcua.Variant({dataType: opcua.DataType.Double, value: value});
            }
        }
    });

    // Rpi Up time in hours
    server.nodeVariable2 = addressSpace.addVariable({
        componentOf: Rpi_info,
        nodeId: "ns=24;b=1020AE",
        browseName: "Up time in hours",
        dataType: "Double",
        minimumSamplingInterval: 1000,
        value: {
            get: function () {
                var value = os.uptime()/60/60;
                return new opcua.Variant({dataType: opcua.DataType.Double, value: value});
            }
        }
    });

    // CPU load in on 15m
    server.nodeVariable3 = addressSpace.addVariable({
        componentOf: Rpi_info,
        nodeId: "ns=25;b=1020AE",
```

```
            browseName: "Load Core 15m",
            dataType: "Double",
            minimumSamplingInterval: 1000,
            value: {
                get: function () {
                    var value = os.loadavg()[2];
                    return new opcua.Variant({dataType: opcua.DataType.Double, value:
                        value});
                }
            }
        });
//------------Add variables---------------END
    }

    //Construct the address space and start the server
    construct_my_address_space(server);
    server.start(function() {
        console.log("Server is now listening ... ( press CTRL+C to stop)");
        console.log("port ", server.endpoints[0].port);
        var endpointUrl = server.endpoints[0].endpointDescriptions()[0].endpointUrl;
        console.log(" the primary server endpoint url is ", endpointUrl );
    });
}
server.initialize(post_initialize);
```

A.3.2 Arduino digital in code

Code A.2: Digital in code for Arduino I/O module, written in Arduino IDE.

```
/*
   Digital In x4
   Created by Sturla on 20/2/16.
   Last updated 25/04/16.
*/
#include <Wire.h> // Include wire library for I2C

int DeviceID = 3;
// Define the pins
const int sensorPin1 = 4;
const int sensorPin2 = 7;
const int sensorPin3 = 8;
const int sensorPin4 = 12;
// Define the Sensor Values
int sensorValue1 = 0;
int sensorValue2 = 0;
int sensorValue3 = 0;
int sensorValue4 = 0;

byte packet[13]; //Bytes to be received
int sum = 0; //Initialize the sum variable
byte byteVector[1]; //command byte from Rpi to identify which port

void setup()
{
  // Define the Pins used
  pinMode(sensorPin1, INPUT);
  pinMode(sensorPin2, INPUT);
  pinMode(sensorPin3, INPUT);
  pinMode(sensorPin4, INPUT);
  Wire.begin(6); // Setting this Arduino to I2C address 6
  Wire.onRequest(requestEvent); // Act when requested
```

```
  Wire.onReceive(receiveEvent); // Receive command byte
  //Serial.begin(9600); // For debugging

  // Build the protocol string with known information
  packet[0] = 1; //Byte Source Device ID
  packet[1] = 3; //Byte Device ID
  packet[5] = 0; //Byte Command
  packet[6] = 0; //Byte3 Message (not used)
  packet[7] = 0; //Byte4 Message (not used)
  packet[8] = 0; //Byte5 Message (not used)
  packet[9] = 0; //Byte6 Message (not used)
  packet[10] = 0; //Byte7 Message (not used)
  packet[11] = 0; //Byte8 Message (not used)
  packet[12] = 1; //Byte Message length
}

void loop() // Main loop
{

}

void requestEvent() // Only run on request from Rpi through I2C
{

  if (byteVector[0] == 1) // If request for port 1
  {
    sensorValue1 = digitalRead(sensorPin1); // Read pin on port 1
    packet[2] = 1;
    packet[3] = byteVector[0];
    packet[4] = sensorValue1;
  }

  else if (byteVector[0] == 2) // If request for port 2
  {
    sensorValue2 = digitalRead(sensorPin2); // Read pin on port 2
    packet[2] = 2;
    packet[3] = byteVector[0];
    packet[4] = sensorValue2;
  }

  else if (byteVector[0] == 3) // If request for port 3
  {
    sensorValue3 = digitalRead(sensorPin3); // Read pin on port 3
    packet[2] = 3;
    packet[3] = byteVector[0];
    packet[4] = sensorValue3;
  }

  else if (byteVector[0] == 4) // If request for port 4
  {
    sensorValue4 = digitalRead(sensorPin4); // Read pin on port 4
    packet[2] = 4;
    packet[3] = byteVector[0];
    packet[4] = sensorValue4;
  }

  //Check sum calculation
  for (int i = 0; i < 13; i++)
  {
    sum += packet[i];
  }
  packet[13] = (sum + 624) % 64; // Remainder
  Wire.write(packet, 14); // Sending the byte string to Rpi
```

```
  sum = 0; // Reset sum
  packet[13] = 0; // Reset check sum
  byteVector[0] = 0; // Reset Source Device ID
}

void receiveEvent(int howMany) // Recieve the command byte (which port) from
    Rpi
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 1; i++) //Read 1 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect byte to byte vector
    }
  }
}
```

A.3.3 Arduino analog in code

Code A.3: Analog in code for Arduino, written in Arduino IDE.

```
/*
   Analog In x4
   Created by Sturla on 1/3/16.
   Last updated 25/04/16.
*/
#include <Wire.h> // Include wire library for I2C

int DeviceID = 4;
// Define the pins
const int sensorPin1 = A0;
const int sensorPin2 = A1;
const int sensorPin3 = A2;
const int sensorPin4 = A3;
// Define the Sensor Values
String sensorValue1;
String sensorValue2;
String sensorValue3;
String sensorValue4;
int StrLength1;
int StrLength2;
int StrLength3;
int StrLength4;

byte packet[13]; //Bytes to be received
int sum; //Initialize the sum variable
byte byteVector[1]; //command byte from Rpi to identify which port

void setup()
{
  Wire.begin(5); // Setting this Arduino to I2C address 5
  Wire.onRequest(requestEvent); // Act when requested
  Wire.onReceive(receiveEvent); // register event
  //Serial.begin(9600); // For debugging
  // Build the protocol string with known information
  packet[0] = 1; //Byte Source Device ID
  packet[1] = 4; //Byte Device ID
  packet[3] = 0; //Byte Command
  packet[8] = 0; //Byte5 Message (not used)
  packet[9] = 0; //Byte6 Message (not used)
```

```
  packet[10] = 0; //Byte7 Message (not used)
  packet[11] = 0; //Byte8 Message (not used)
}

void loop() // Main loop
{

}

void requestEvent() // Only run on request from Rpi through I2C
{

  if (byteVector[0] == 1) // If request for port 1
  {
    sensorValue1 = String(analogRead(sensorPin1)); // Read pin on port 1
    StrLength1 = sensorValue1.length();
    packet[2] = byteVector[0];
    if (StrLength1 == 1) // If the output value is 1 digit long
    {
      packet[4] = sensorValue1.substring(0, 1).toInt();
      packet[5] = 0;
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength1 == 2) // If the output value is 2 digit long
    {
      packet[4] = sensorValue1.substring(0, 1).toInt();
      packet[5] = sensorValue1.substring(1, 2).toInt();
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength1 == 3) // If the output value is 3 digit long
    {
      packet[4] = sensorValue1.substring(0, 1).toInt();
      packet[5] = sensorValue1.substring(1, 2).toInt();
      packet[6] = sensorValue1.substring(2, 3).toInt();
      packet[7] = 0;
    }
    else if (StrLength1 == 4) // If the output value is 4 digit long
    {
      packet[4] = sensorValue1.substring(0, 1).toInt();
      packet[5] = sensorValue1.substring(1, 2).toInt();
      packet[6] = sensorValue1.substring(2, 3).toInt();
      packet[7] = sensorValue1.substring(3, 4).toInt();
    }
    packet[12] = StrLength1;
  }

  else if (byteVector[0] == 2) // If request for port 2
  {
    sensorValue2 = String(analogRead(sensorPin2)); // Read pin on port 2
    StrLength2 = sensorValue2.length();
    packet[2] = byteVector[0];
    if (StrLength2 == 1) // If the output value is 1 digit long
    {
      packet[4] = sensorValue2.substring(0, 1).toInt();
      packet[5] = 0;
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength2 == 2) // If the output value is 2 digit long
    {
      packet[4] = sensorValue2.substring(0, 1).toInt();
```

```
      packet[5] = sensorValue2.substring(1, 2).toInt();
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength2 == 3) // If the output value is 3 digit long
    {
      packet[4] = sensorValue2.substring(0, 1).toInt();
      packet[5] = sensorValue2.substring(1, 2).toInt();
      packet[6] = sensorValue2.substring(2, 3).toInt();
      packet[7] = 0;
    }
    else if (StrLength2 == 4) // If the output value is 4 digit long
    {
      packet[4] = sensorValue2.substring(0, 1).toInt();
      packet[5] = sensorValue2.substring(1, 2).toInt();
      packet[6] = sensorValue2.substring(2, 3).toInt();
      packet[7] = sensorValue2.substring(3, 4).toInt();
    }
    packet[12] = StrLength2;
  }

  else if (byteVector[0] == 3) // If request for port 3
  {
    sensorValue3 = String(analogRead(sensorPin3)); // Read pin on port 3
    StrLength3 = sensorValue3.length();
    packet[2] = byteVector[0];
    if (StrLength3 == 1) // If the output value is 1 digit long
    {
      packet[4] = sensorValue3.substring(0, 1).toInt();
      packet[5] = 0;
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength3 == 2) // If the output value is 2 digit long
    {
      packet[4] = sensorValue3.substring(0, 1).toInt();
      packet[5] = sensorValue3.substring(1, 2).toInt();
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength3 == 3) // If the output value is 3 digit long
    {
      packet[4] = sensorValue3.substring(0, 1).toInt();
      packet[5] = sensorValue3.substring(1, 2).toInt();
      packet[6] = sensorValue3.substring(2, 3).toInt();
      packet[7] = 0;
    }
    else if (StrLength3 == 4) // If the output value is 4 digit long
    {
      packet[4] = sensorValue3.substring(0, 1).toInt();
      packet[5] = sensorValue3.substring(1, 2).toInt();
      packet[6] = sensorValue3.substring(2, 3).toInt();
      packet[7] = sensorValue3.substring(3, 4).toInt();
    }
    packet[12] = StrLength3;
  }

  else if (byteVector[0] == 4) // If request for port 4
  {
    sensorValue4 = String(analogRead(sensorPin4)); // Read pin on port 4
    StrLength4 = sensorValue4.length();
    packet[2] = byteVector[0];
    if (StrLength4 == 1) // If the output value is 1 digit long
```

```
    {
      packet[4] = sensorValue4.substring(0, 1).toInt();
      packet[5] = 0;
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength4 == 2) // If the output value is 2 digit long
    {
      packet[4] = sensorValue4.substring(0, 1).toInt();
      packet[5] = sensorValue4.substring(1, 2).toInt();
      packet[6] = 0;
      packet[7] = 0;
    }
    else if (StrLength4 == 3) // If the output value is 3 digit long
    {
      packet[4] = sensorValue4.substring(0, 1).toInt();
      packet[5] = sensorValue4.substring(1, 2).toInt();
      packet[6] = sensorValue4.substring(2, 3).toInt();
      packet[7] = 0;
    }
    else if (StrLength4 == 4) // If the output value is 4 digit long
    {
      packet[4] = sensorValue4.substring(0, 1).toInt();
      packet[5] = sensorValue4.substring(1, 2).toInt();
      packet[6] = sensorValue4.substring(2, 3).toInt();
      packet[7] = sensorValue4.substring(3, 4).toInt();
    }
    packet[12] = StrLength4;
  }

  //Check sum calculation
  for (int i = 0; i < 13; i++)
  {
    sum += packet[i];
  }
  packet[13] = (sum + 624) % 64; // Remainder
  Wire.write(packet, 14); // Sending byte the string to Rpi
  sum = 0; // Reset sum
  packet[13] = 0; // Reset check sum
  byteVector[0] = 0; // Reset Source Device ID
}

void receiveEvent(int howMany) // Recieve the command byte (which port) from
    Rpi
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 1; i++) //Read 1 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect byte to byte vector
    }
  }
}
```

A.3.4 Arduino digital out code

Code A.4: Digital out code for Arduino I/O module, written in Arduino IDE.

```
/*
   Digital Out x4
```

```
  Created by Sturla on 26/2/16.
  Last updated 25/04/16.
*/
//Note that all printing to Serial in this program is only for debugging purposes

#include <Wire.h> // Include wire library for I2C

int DeviceID = 2; // Define the Device ID

// Define the pins
const int outputPin1 = 4;
const int outputPin2 = 5;
const int outputPin3 = 6;
const int outputPin4 = 9;

byte byteVector[15]; // Define the bytes vector

// Define the message strings for further processing
String MessageByte1;
String MessageByte2;
String MessageByte3;
String MessageByte4;
String MessageByte5;
String MessageByte6;
String MessageByte7;
String MessageByte8;

int s; // Define the check sum

void setup()
{
  // Define the Pins used
  pinMode(outputPin1, OUTPUT);
  pinMode(outputPin2, OUTPUT);
  pinMode(outputPin3, OUTPUT);
  pinMode(outputPin4, OUTPUT);
  Wire.begin(7); // Setting this Arduino to I2C address 7
  Wire.onReceive(receiveEvent); // Act when recieving
  Serial.begin(9600); // For debugging
}

void loop() // Main loop
{

  //If the right Device ID is recognized
  if (byteVector[1] == DeviceID)
  {
    //Sum Checksum Error
    for (int i = 0; i < 13; i++)
    {
      s += byteVector[i] + 48;
    }

    Serial.print("sum = ");
    Serial.println(s);
    s = s % 64; // Remainder
    Serial.print("Checksum = ");
    Serial.println(s);
    Serial.println("_______");
    byteVector[1] = 0; // Reset DeviceID
  }

```

```
  //Checksum Error Detection, senders checksum == to calculated checksum
  if (byteVector[13] == s)
  {
    Serial.println("ooooooooo");
    Serial.println("No Error Detected!");
    Serial.println("ooooooooo");
    Serial.println(" ");
    Serial.println(" ");
    Serial.println(" ");
    byteVector[13] = 1;

    // Byte 2 indicates which pin
    if (byteVector[2] == 1) // If OutputPin 1
    {
      if (byteVector[4] == 0) // If byte = 0 then LOW
      {
        digitalWrite(outputPin1, LOW);
      }
      else if (byteVector[4] == 1) // If byte = 1 then HIGH
      {
        digitalWrite(outputPin1, HIGH);
      }
    }

    else if (byteVector[2] == 2) // If OutputPin 2
    {
      if (byteVector[4] == 0)
      {
        digitalWrite(outputPin2, LOW);
      }
      else if (byteVector[4] == 1)
      {
        digitalWrite(outputPin2, HIGH);
      }
    }

    else if (byteVector[2] == 3) // If OutputPin 3
    {
      if (byteVector[4] == 0)
      {
        digitalWrite(outputPin3, LOW);
      }
      else if (byteVector[4] == 1)
      {
        digitalWrite(outputPin3, HIGH);
      }
    }

    else if (byteVector[2] == 4) // If OutputPin 4
    {
      if (byteVector[4] == 0)
      {
        digitalWrite(outputPin4, LOW);
      }
      else if (byteVector[4] == 1)
      {
        digitalWrite(outputPin4, HIGH);
      }
    }
  }
  s = 0; // Reset check sum
}
```

```
void receiveEvent(int howMany) // Only run on request from Rpi through I2C
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 15; i++) //Read 14 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect bytes to byte vector
    }
  }
  /* For debugging
     Serial.println(byteVector[0]); //Byte Source Device ID
     Serial.println(byteVector[1]); //Byte Device ID
     Serial.println(byteVector[2]); //Byte Port
     Serial.println(byteVector[3]); //Byte Command
     Serial.println(byteVector[4]); //Byte1 Message
     Serial.println(byteVector[5]); //Byte2 Message
     Serial.println(byteVector[6]); //Byte3 Message
     Serial.println(byteVector[7]); //Byte4 Message
     Serial.println(byteVector[8]); //Byte5 Message
     Serial.println(byteVector[9]); //Byte6 Message
     Serial.println(byteVector[10]); //Byte7 Message
     Serial.println(byteVector[11]); //Byte8 Message
     Serial.println(byteVector[12]); //Byte Message length
     Serial.println(byteVector[13]); //Byte Checksum
  */
}
```

A.3.5 Arduino analog out code

Code A.5: Analog OUT code for Arduino, written in Arduino IDE.

```
/*
   Analog Out x2
   Created by Sturla on 6/3/16.
   Last updated 25/04/16.
*/
//Note that all printing to Serial in this program is only for debugging
    purposes

#include <Wire.h> // Include wire library for I2C
#include <Adafruit_MCP4725.h> // Include MCP4725-DAC library

Adafruit_MCP4725 dac; // Define Dac 1
Adafruit_MCP4725 dac1; // Define Dac 2
#define MCP4725_ADDR 0x62 // Define Dac 1 address
#define MCP4725_ADDR2 0x63 // Define Dac 2 address

int DeviceID = 1; // Define the Device ID
byte byteVector[15]; // Define the bytes vector
String value;
String value2;
int messageInt = 0;

// Define the message strings for further processing
String MessageByte1;
String MessageByte2;
String MessageByte3;
String MessageByte4;
String MessageByte5;
String MessageByte6;
```

```
String MessageByte7;
String MessageByte8;

int s; // Define the Checsum error
int valueInt;
void setup(void)
{
  Wire.begin(8); // Setting this Arduino to I2C address 8
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600); // For debugging
}

void loop(void)  // Main loop
{

  //If the right Device ID is recognized
  if (byteVector[1] == DeviceID)
  {
    //Sum Checksum Error
    for (int i = 0; i < 13; i++)
    {
      s += byteVector[i] + 48;
    }

    Serial.print("sum = ");
    Serial.println(s);
    s = s % 64; // Remainder
    Serial.print("Checksum = ");
    Serial.println(s);
    Serial.println("_______");
    byteVector[1] = 0; // Reset DeviceID
  }

  //Checksum Error Detection, senders checksum == to calculated checksum
  if (byteVector[13] == s)
  {
    Serial.println("ooooooooo");
    Serial.println("No Error Detected!");
    Serial.println("ooooooooo");
    Serial.println(" ");
    Serial.println(" ");
    Serial.println(" ");
    byteVector[13] = 1;

    // If the identifier byte is 1 then change output for Dac 1
    if (byteVector[2] == 1)
    {
      dac.begin(0x62); // DAC one begin

      // Build the byte string
      MessageByte1 = String(byteVector[4]);
      MessageByte2 = String(byteVector[5]);
      MessageByte3 = String(byteVector[6]);
      MessageByte4 = String(byteVector[7]);
      MessageByte5 = String(byteVector[8]);
      MessageByte6 = String(byteVector[9]);
      MessageByte7 = String(byteVector[10]);
      MessageByte8 = String(byteVector[11]);

      if (byteVector[12] == 1) // If one digit number
      {
        MessageByte1 = String(byteVector[4]);
        value =  String(MessageByte1);
```

```
    }
    else if (byteVector[12] == 2) // If two digit number
    {
      MessageByte1 = String(byteVector[4]);
      MessageByte2 = String(byteVector[5]);
      value =  String(MessageByte1 + MessageByte2);
    }
    else if (byteVector[12] == 3) // If three digit number
    {
      MessageByte1 = String(byteVector[4]);
      MessageByte2 = String(byteVector[5]);
      MessageByte3 = String(byteVector[6]);
      value =  String(MessageByte1 + MessageByte2 + MessageByte3);
    }
    else if (byteVector[12] == 4) // If four digit number
    {
      MessageByte1 = String(byteVector[4]);
      MessageByte2 = String(byteVector[5]);
      MessageByte3 = String(byteVector[6]);
      MessageByte4 = String(byteVector[7]);
      value =  String(MessageByte1 + MessageByte2 + MessageByte3 + MessageByte4);
    }

    valueInt = value.toInt();

    if ((valueInt <= 4095) && (valueInt >= 0)) // Only allowed values from 0-4095 (12bit)
    {
      dac.setVoltage(valueInt, false); // Set Value to DAC 1
      delay(10);

      Serial.println("Value to DAC: " + String(valueInt));
      byteVector[2] = 0; // Reset identifier
      valueInt = 0;
      value = "";
    }
  }

  // If the identifier byte is 2 then change output for Dac 2
  if (byteVector[2] == 2)
  {
    dac1.begin(0x63); // DAC one begin

    // Build the byte string
    MessageByte1 = String(byteVector[4]);
    MessageByte2 = String(byteVector[5]);
    MessageByte3 = String(byteVector[6]);
    MessageByte4 = String(byteVector[7]);
    MessageByte5 = String(byteVector[8]);
    MessageByte6 = String(byteVector[9]);
    MessageByte7 = String(byteVector[10]);
    MessageByte8 = String(byteVector[11]);

    if (byteVector[12] == 1) // If one digit number
    {
      MessageByte1 = String(byteVector[4]);
      value =  String(MessageByte1);
    }
    else if (byteVector[12] == 2) // If two digit number
    {
      MessageByte1 = String(byteVector[4]);
      MessageByte2 = String(byteVector[5]);
```

```
        value =  String(MessageByte1 + MessageByte2);
      }
      else if (byteVector[12] == 3) // If three digit number
      {
        MessageByte1 = String(byteVector[4]);
        MessageByte2 = String(byteVector[5]);
        MessageByte3 = String(byteVector[6]);
        value =  String(MessageByte1 + MessageByte2 + MessageByte3);
      }
      else if (byteVector[12] == 4) // If four digit number
      {
        MessageByte1 = String(byteVector[4]);
        MessageByte2 = String(byteVector[5]);
        MessageByte3 = String(byteVector[6]);
        MessageByte4 = String(byteVector[7]);
        value =  String(MessageByte1 + MessageByte2 + MessageByte3 + MessageByte4);
      }

      valueInt = value.toInt(); // String to Int

      if ((valueInt <= 4095) && (valueInt >= 0)) // Only allowed values from 0-4095 (12bit)
      {
        dac1.setVoltage(valueInt, false); // Set Value to DAC 2
        delay(10);

        Serial.println("Value to DAC: " + String(valueInt));
        byteVector[2] = 0; // Reset identifier
        valueInt = 0;
        value = "";
      }
    }
  }
  s = 0; // Reset check sum
}

void receiveEvent(int howMany) // Only receive on request from Rpi through I2C
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 15; i++) //Read 14 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect bytes to byte vector
    }
  }
  /* For debugging
    Serial.println(byteVector[0]); //Byte Source Device ID
    Serial.println(byteVector[1]); //Byte Device ID
    Serial.println(byteVector[2]); //Byte Port
    Serial.println(byteVector[3]); //Byte Command
    Serial.println(byteVector[4]); //Byte1 Message
    Serial.println(byteVector[5]); //Byte2 Message
    Serial.println(byteVector[6]); //Byte3 Message
    Serial.println(byteVector[7]); //Byte4 Message
    Serial.println(byteVector[8]); //Byte5 Message
    Serial.println(byteVector[9]); //Byte6 Message
    Serial.println(byteVector[10]); //Byte7 Message
    Serial.println(byteVector[11]); //Byte8 Message
    Serial.println(byteVector[12]); //Byte Message length
    Serial.println(byteVector[13]); //Byte Checksum
```

```
  */
}
```

A.3.6 Arduino pulse timing measurement code

Code A.6: Measurement code for Arduino that times the duration between pulses received, written in Arduino IDE.

```
/*
   For measuring pulse timing on pin 2
   Created by Sturla on 5/04/16.
   Last updated 25/04/16.
*/

const int MeasurePin = 2;  // Initialize the input pin

long startTime; // The value returned from millis when signal comes in
long duration;  // Variable to store the duration

void setup()
{
  pinMode(MeasurePin, INPUT);
  digitalWrite(MeasurePin, HIGH);
  Serial.begin(9600);
}

void loop() // Main loop
{
  if (digitalRead(MeasurePin) == LOW) // If state change is LOW
  {
    startTime = millis(); // Start the timer
    while (digitalRead(MeasurePin) == LOW) // While state is LOW
    {
      // wait for state change to HIGH
    }
    long duration = millis() - startTime; // Store the duration
    Serial.println(duration); // Print duration to Serial
  }
}
```

A.3.7 Arduino analog out code to use with the Arduino measurement code (for checksum error)

Code A.7: Code for Arduino Analog out to use with the Code A.3.6. It is programmed to output 5 V 50 ms pulse from the DAC when there is an checksum error, written in Arduino IDE

```
/*
   Analog Out x2, for checsum error measurements
   Created by Sturla on 20/3/16.
   Last updated 25/04/16.
*/
#include <Wire.h> // Include wire library for I2C
#include <Adafruit_MCP4725.h> // Include MCP4725-DAC library

Adafruit_MCP4725 dac; // Define Dac 1
Adafruit_MCP4725 dac1; // Define Dac 2
#define MCP4725_ADDR 0x62 // Define Dac 1 address
```

```
#define MCP4725_ADDR2 0x63 // Define Dac 2 address

int DeviceID = 1;
byte byteVector[15]; // Define the bytes vector

int s; // Define the Checsum error
int valueInt;

double counter1 = 0; // Counts how many requests have no errors
double counter2 = 0;
int CheckSumPassed = 1;
double checksumerrorperc = 0;
//double checksumerrorperc = 0; // Error percentage
int TurnOn = 1; // variable for turning on output pin (LED)

void setup(void)
{
  Wire.begin(8); // Setting this Arduino to I2C address 8
  Wire.onReceive(receiveEvent); // register event
  //Serial.begin(9600); // For debugging
}

void loop(void)  // Main loop
{

  // If there is checksum error
  if (TurnOn == 1)
  {
    dac1.begin(0x63);
    dac1.setVoltage(4095, false); //Set DAC to 5V
    delay(50); // Let the pulse be 50ms
    dac1.setVoltage(0, false); //Set DAC to 0V
    TurnOn = 0;
    //Serial.println(checksumerrorperc, 6);
    //Serial.println(counter1, 6);
    counter1 = 0; // Reset counter
  }
}

void receiveEvent(int howMany) // Only run on request from Rpi through I2C
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 14; i++) //Read 14 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect bytes to byte vector
    }
  }

  //Sum Checksum Error
  for (int i = 0; i < 13; i++)
  {
    s += byteVector[i] + 48;
  }

  s = s % 64; // Remainder
  counter1 = counter1 + 1;

  if ((byteVector[13] != s))
  {
    checksumerrorperc = 1 / counter1;
    counter2 = counter2 + 1;
```

```

    TurnOn = 1; // turn on DAC (LED)

  }
  s = 0; // Reset check sum
}
```

A.3.8 Arduino digital out code to use with the Arduino Measurement code (for checksum error)

Code A.8: Code for Arduino Digital out to use with the Code A.3.6. It programmed to output 5 V 50 ms signal when there is an checksum error, written in Arduino IDE

```
/*
   Digital Out x4, for checksum error measurements
   Created by Sturla on 10/4/16.
   Last updated 25/04/16.
*/
#include <Wire.h> // Include wire library for I2C

int DeviceID = 2; // Define the Device ID

// Define the output pins
const int outputPin1 = 4;
const int outputPin2 = 5;
const int outputPin3 = 6;
const int outputPin4 = 9;
byte byteVector[15]; // Define the bytes vector

// Define the message strings for further processing
String MessageByte1;
String MessageByte2;
String MessageByte3;
String MessageByte4;
String MessageByte5;
String MessageByte6;
String MessageByte7;
String MessageByte8;

int s; // Define the Checsum error
int valueInt;

double counter1 = 0; // Counts how many requests have no errors
double counter2 = 0;
int CheckSumPassed = 1;
//double checksumerrorperc = 0; // Error percentage
int TurnOn = 1; // variable for turning on output pin (LED)
int counterSwitch = 0; // for switching output pin on and off

void setup()
{
  // Define the Pins used
  pinMode(outputPin1, OUTPUT);
  pinMode(outputPin2, OUTPUT);
  pinMode(outputPin3, OUTPUT);
  pinMode(outputPin4, OUTPUT);
  Wire.begin(7); // Setting this Arduino to I2C address 7
  Wire.onReceive(receiveEvent); // Act when recieving
  //Serial.begin(9600); // For debugging
}
```

```

void loop() // Main loop
{

  // If there is checksum error
  if (TurnOn == 1)
  {
    TurnOn = 0; // Reset TurnOn
    counter1 = 0; // Reset counter1
    counterSwitch = counterSwitch + 1;

    // If last time it was LOW then HIGH and vice versa
    if (counterSwitch == 1)
    {
      digitalWrite(outputPin1, HIGH);
    }
    if (counterSwitch == 2)
    {
      digitalWrite(outputPin1, LOW);
      counterSwitch = 0; // Reset counterSwitch
    }
    delay(50); // Let the pulse be 50ms
  }
}

void receiveEvent(int howMany) // Only receive on request from Rpi through
    I2C
{
  while ( Wire.available()) // While the bytes keep coming
  {
    for (int i = 0; i < 15; i++) //Read 14 bytes
    {
      byte b = Wire.read(); // Read from Rpi through I2C
      byteVector[i] = b; // Collect bytes to byte vector
    }
  }

  // Sum check sum Error
  for (int i = 0; i < 13; i++)
  {
    s += byteVector[i] + 48;
  }
  s = s % 64; // Remainder
  counter1 = counter1 + 1;

  // If there is a check sum error!
  if ((byteVector[13] != s))
  {
    //checksumerrorperc = 1 / counter1; // Error percentage
    counter2 = counter2 + 1;
    TurnOn = 1; // turn on output pin (LED)
  }
  s = 0; // Reset check sum
}
```

A.3.9 Arduino code for power cut-off experiment

Code A.9: Arduino code for the power cut-off experiment that tests the durability of the SD card in Raspberry Pi 2.

```
/*
   Digital Out x4, for checksum error measurements
   Created by Sturla on 30/4/16.
   Last updated 16/05/16.
*/

// Define the output pin
const int outputPin1 = 4;

void setup()
{
  // Define Pin used
  pinMode(outputPin1, OUTPUT);
}

void loop() // Main loop
{
  digitalWrite(outputPin1, HIGH); //Turn on (HIGH) for 4 min
  delay(1000*60*4);
  digitalWrite(outputPin1, LOW); //Turn off (LOW) for 1 min
  delay(1000*60*1);
}
```

A.3.10 LabVIEW client code

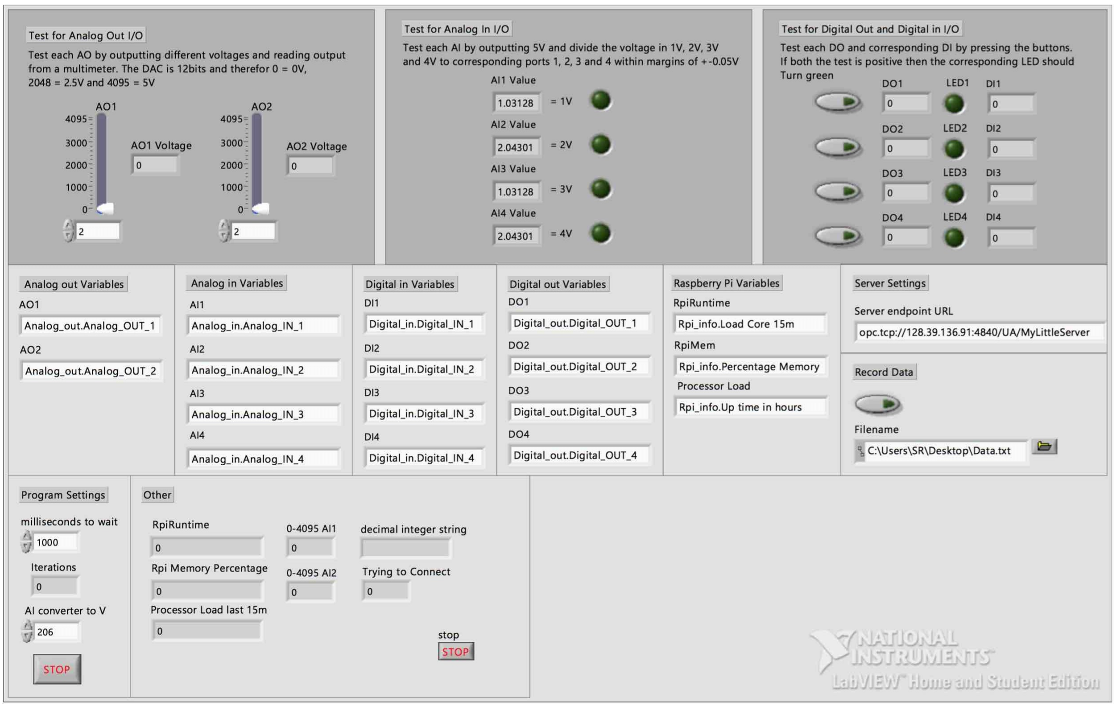


Figure A.1: Front panel GUI of the LabVIEW OPC-UA client used for testing the prototype. It shows three main windows that are for controlling AO, reading AI and reading/writing to DI/DO.

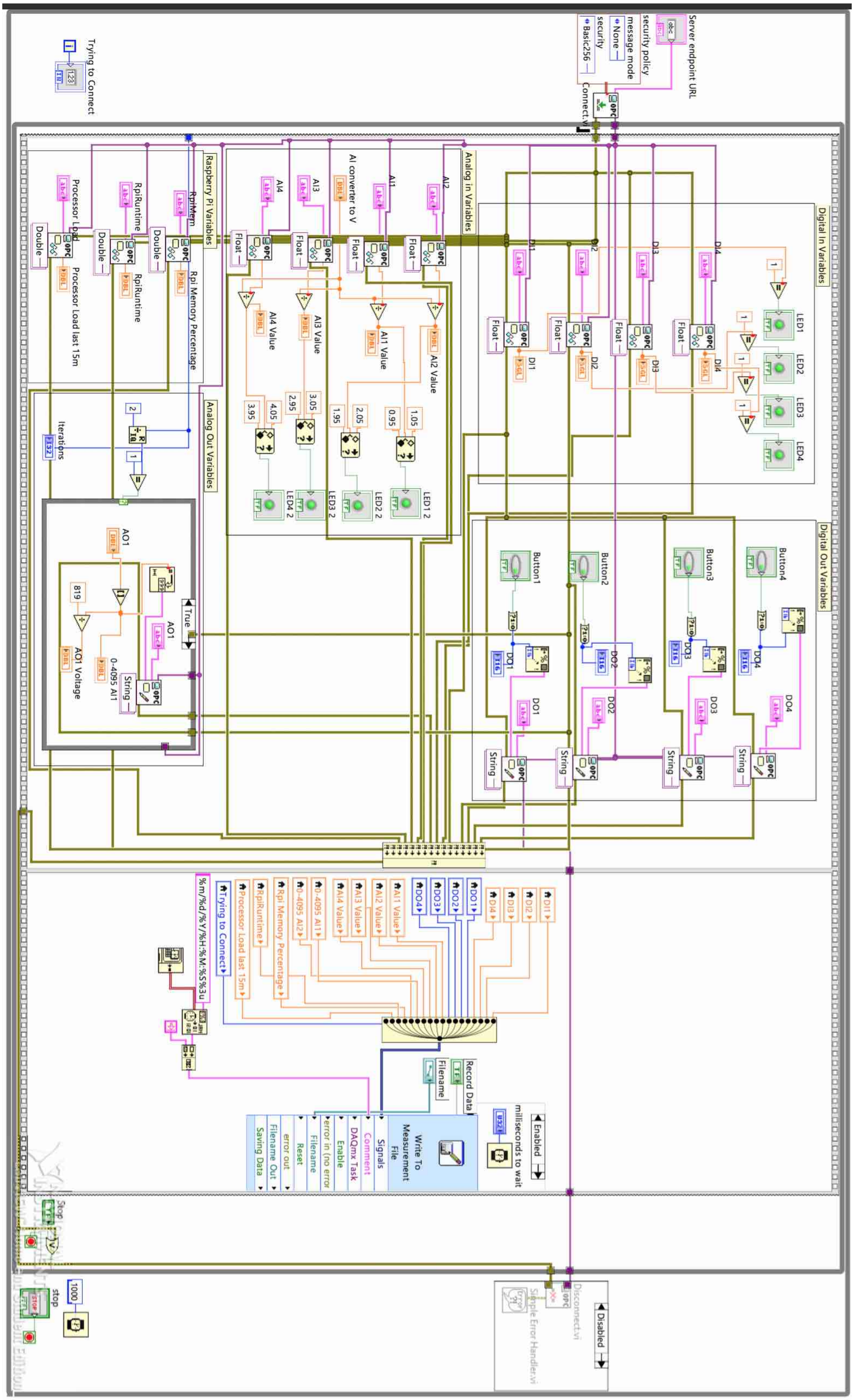


Figure A.2: LabVIEW OPC-UA client code used for testing the prototype. The case structure inside the main loop is divided in two parts. First part is where the OPC-UA variables are defined and the second one is where the variables are logged to a file.

Høgskolen
i Sørøst-Norge

Test Plan Document

Author: Sturla Runarsson

Test Plan for PLC Prototype

Developed in the thesis: "Open Source Hardware and Software Alternative to Industrial PLC"

University College of Southeast Norway
Faculty of Technology

Contents

1 Introduction

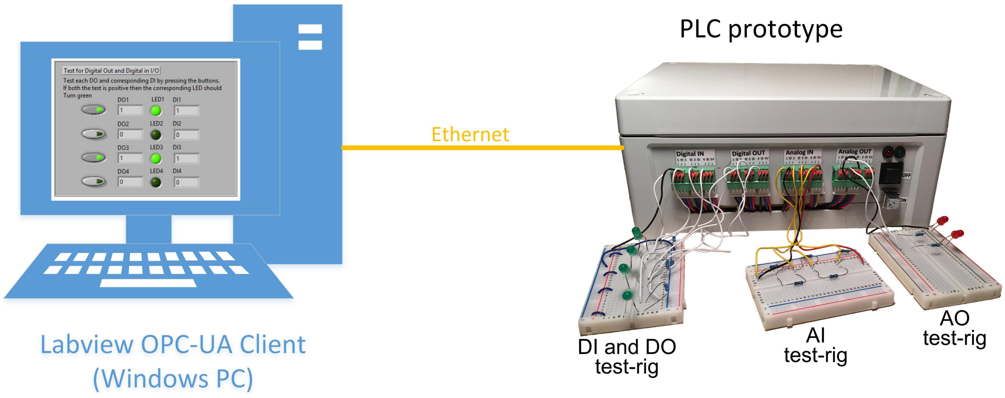


Figure 1: Overview of the Programmable Logic Controller (PLC) prototype with all three test-rigs presented and connection to a Personal Computer (PC) running Open Platform Communications Unified Architecture (OPC-UA) client via Ethernet.

This test plan document was written for the PLC prototype developed in the Master's Thesis "Open source hardware and software alternative to industrial PLC". It was written at University College of Southeast Norway in spring 2016. This document should be used to test exact hardware copies of the prototype. These tests make sure that modules are wired correctly and that signals behave accordingly and within margins on all ports. See Figure 1 for overview of the setup which includes a LabVIEW OPC-UA client on PC running on windows. The LabVIEW OPC-UA client developed by the author is expected to be used with this test plan. It is connected to the prototype via Ethernet and three test-rigs are then connected to the ports on the prototype. The Note that "test-rig" in this context means the physical breadboards and components that plug into the input/output ports on the prototype.

2 System description

2.1 Prototype hardware and wiring

The system consists of a power module, OPC-UA module and four I/O modules which include Digital In (DI), Digital Out (DO), Analog In (AI), Analog Out (AO). Table 1 lists the pin connections on the I/O modules and is intended for reference with the wiring diagram on Figure 2 where all hardware modules and smaller components are shown. Note that all hardware on Figure 2 is within the prototype housing. Wiring diagrams, software structure and test-rig setup for all test cases are shown in next sections.

Table 1: Pin connections on the Arduino I/O modules are listed, including digital in, digital out, analog in and analog out

Arduino Digital in Module	**Pin**	**Arduino Digital out Module**	**Pin**	**Arduino Analog in Module**	**Pin**	**Arduino Analog out Module**	**Pin**	**Raspberry Pi OPC-UA Server Module**	**Pin**
Port 1	4	**Port 1**	4	**Port 1**	A0	**12 V**	Vin	**Shutdown Button**	GPIO 19
Port 2	7	**Port 2**	5	**Port 2**	A1	**I2C bus**	SDA	**Start Button**	SCL
Port 3	8	**Port 3**	6	**Port 3**	A2	**I2C bus**	SCL	**Green LED**	GPIO 5
Port 4	12	**Port 4**	9	**Port 4**	A3	**-**	Ground	**Red LED**	4
12 V	Vin	**12 V**	Vin	**12 V**	Vin			**5 V**	Vin
I2C bus	SDA	**I2C bus**	SDA	**I2C bus**	SDA			**I2C bus**	SDA
I2C bus	SCL	**I2C bus**	SCL	**I2C bus**	SCL			**I2C bus**	SCL
-	Ground	**-**	Ground	**-**	Ground			**-**	Ground

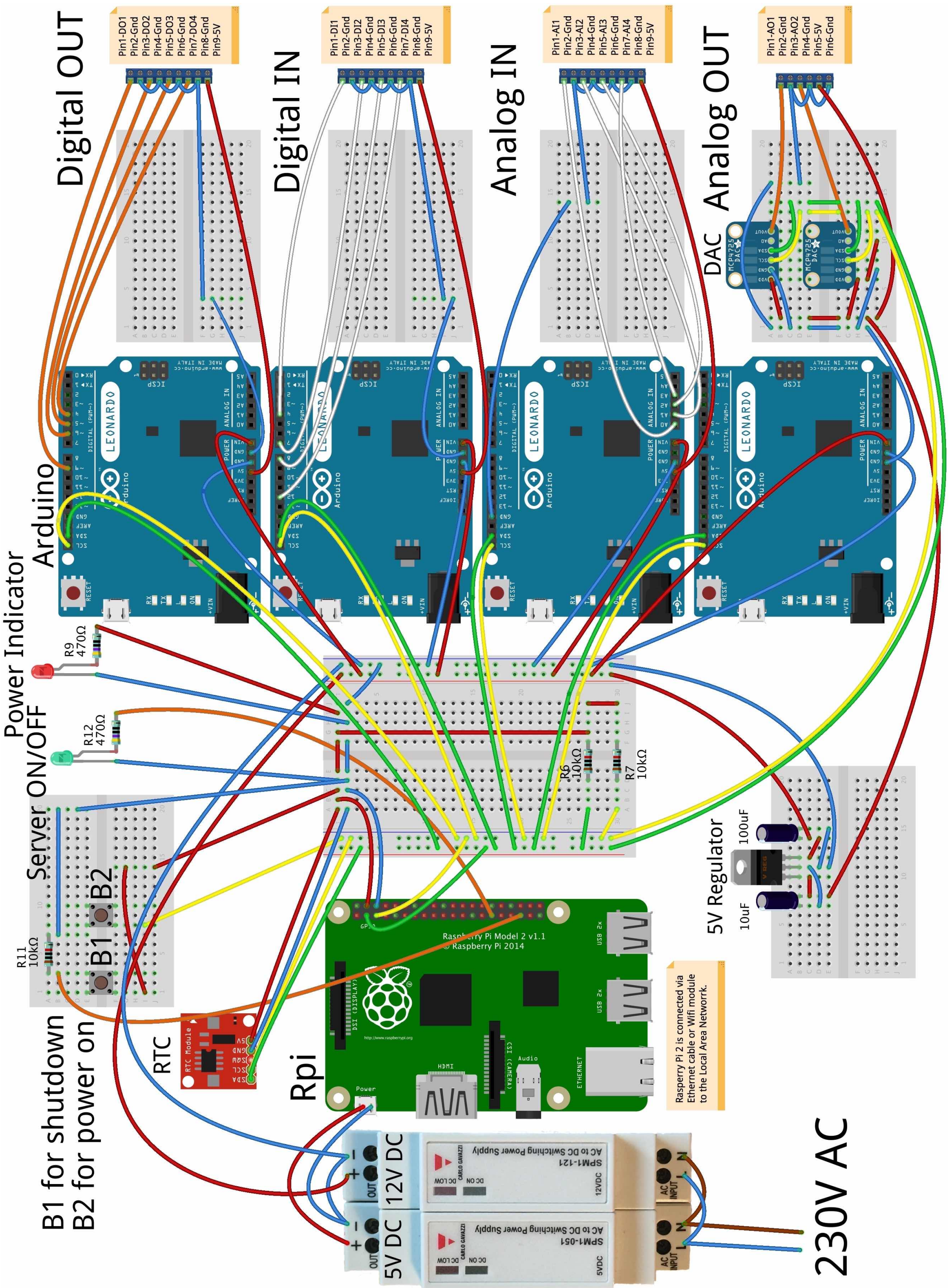


Figure 2: Schematic overview of the whole system with all connections within the prototype housing, including Raspberry Pi 2 module, four Arduino Leonardo I/O, 5V and 12V power supply, real time clock and two DAC's.

2.2 Test setup for digital in / digital out

Both the digital in module and digital out module are tested together. digital in module is tested with the outputs from a digital out module. The software flow diagram can be seen on Figure 3 where the testing of one input port and one output port is shown. The functionality is then extended for four input and output ports.

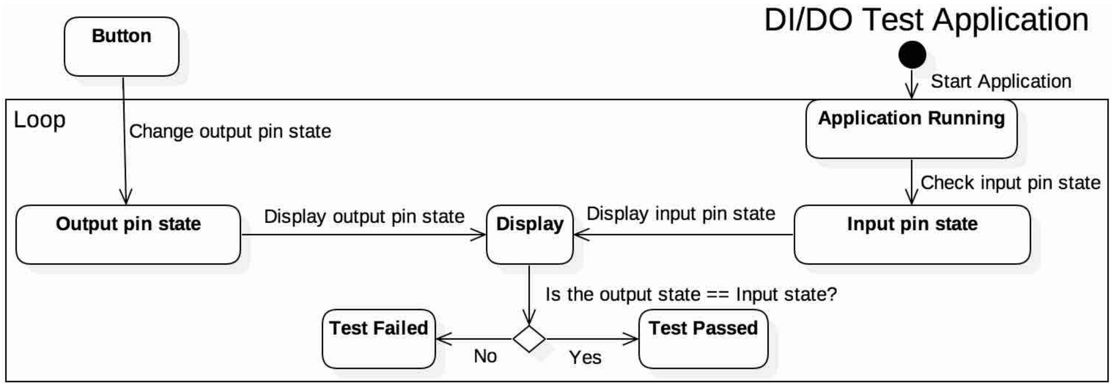
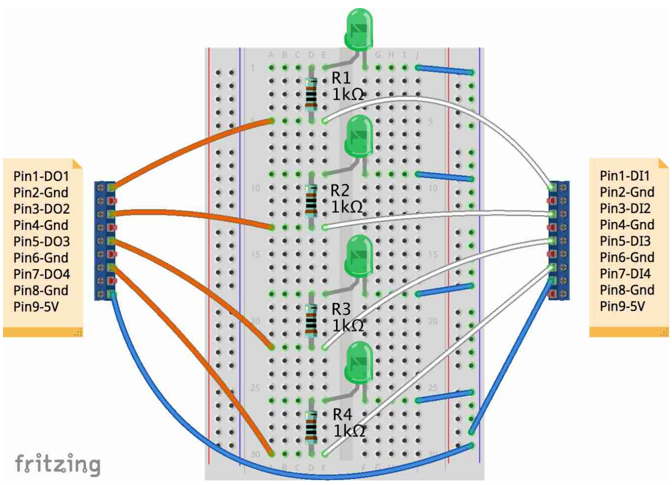
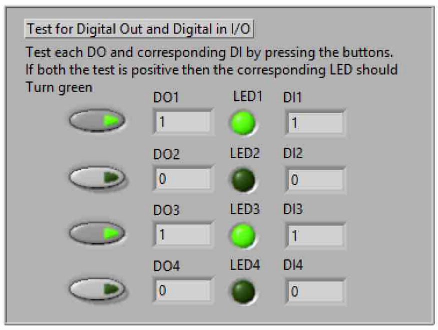


Figure 3: Simple flow diagram of the software that is responsible for testing digital in and digital out modules. It shows state change for one output pin and one input pin which is compared on the display. The program can then be extended for four pins.

The test-rig and test-program Graphical User Interface (GUI) can be seen on Figure 4. The four input ports are tested with the output ports by pressing a corresponding button on the LabVIEW test program. If the digital output port is working correctly then corresponding digital in port will respond with a green LED. If, for example, DO1 is LOW(0) and then turned to HIGH(1) then DI1 should indicate the value 1 with a green LED. If DI1 will not respond to a change in DO1 then the test failed.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 4: Test-rig where digital in ports are tested with digital out ports in LabVIEW . HIGH and LOW signals are outputted and checked whether corresponding signals appear on the inputs.

2.3 Test setup for analog in

The software flow diagram can be seen on Figure 5 where the 10-bit Analog to Digital Converter (ADC) value, which is an integer between 0-1023, is measured and converted to voltages. The specific voltage is then compared to predetermined range that is considered acceptable.

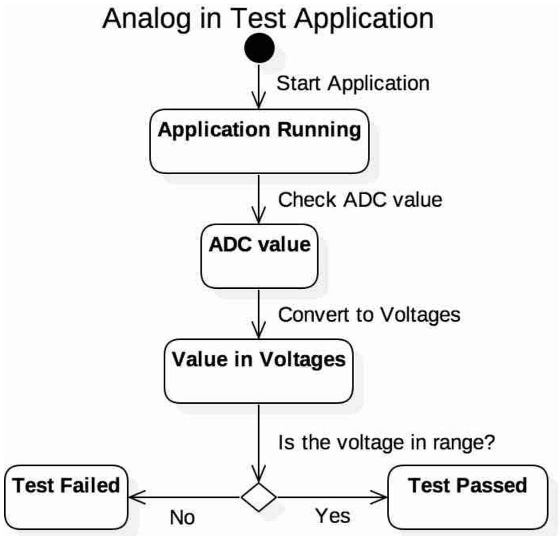
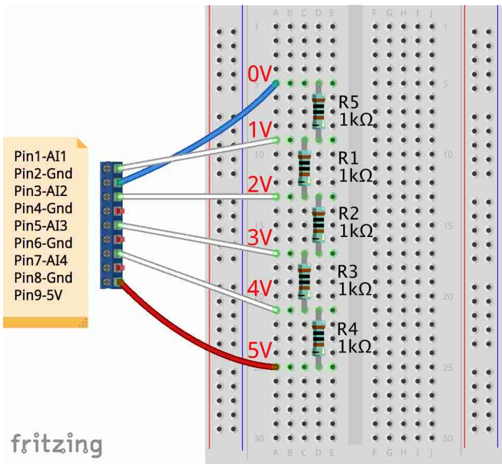
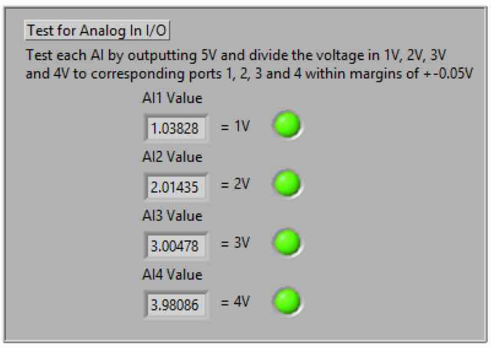


Figure 5: Simple flow diagram of the software that is responsible for testing AI modules. It acquires the ADC value, converts it to voltage and compares it to the specific, allowed range. This program is then extended to four ports.

The test-trig and test-program GUI can be seen on on Figure 6. The test-rig has 5 V supplied that is divided with four 1 kΩ resistors which results in 1 V, 2 V, 3 V and 4 V to the corresponding ports 1, 2, 3 and 4. The LabVIEW program approves with a green LED if the readings are within 0.05V from the correct values, else the test fails.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 6: Test-rig for making sure the analog in module is reading the correct voltages, it reads 4 V, 3 V, 2 V and 1 V on the relative ports.

2.4 Test setup for analog out

The software flow diagram can be seen on Figure 7 where user sets specific voltage to a DAC. The voltage is then measured with a multimeter and compared to predetermined range that is considered acceptable.

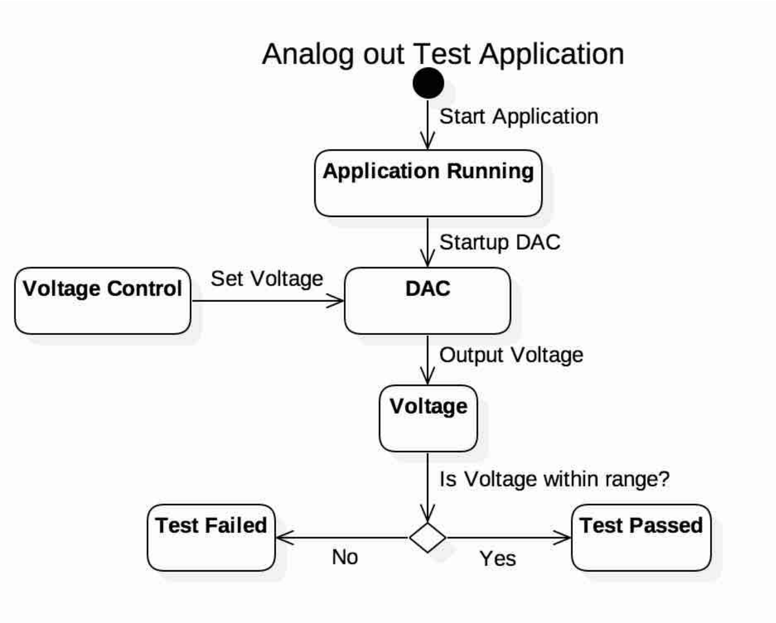
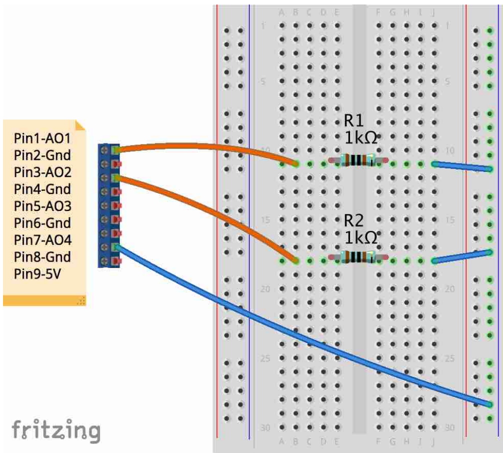
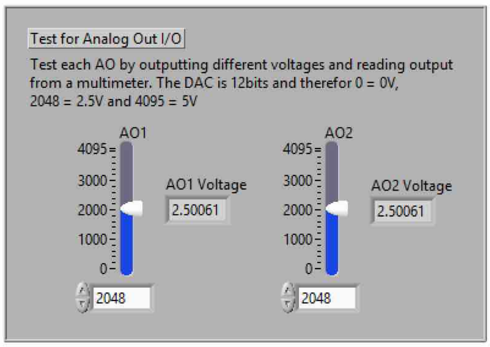


Figure 7: Simple flow diagram of the software that is responsible for testing AO modules. The user sets the output voltage and then measures the output with a multimeter and compare.

The test-rig and test-program GUI can be seen on Figure 8. It makes sure that the analog out module is working properly. Multimeter is needed to measure the voltage drop over the 1 KΩ resistors and compare it to the output command in the LabVIEW test program.



(a) Test-rig.



(b) LabVIEW GUI.

Figure 8: Test-rig for making sure the analog on module is outputting the correct voltages. Multimeter is used to check the voltage drop over the resistors and then compared to the controlled output.

2.5 Test plan checklist

Check list on Table 2 is supposed to be filled out by the technician that tests the prototype. It is expected that this list will be filled out at the same time as the prototype is tested. After completion the technician should sign the test plan document in Section 3.

Table 2: Prototype Test Plan table that is used with the test applications. It is intended for a technician to fill in as a check list.

Modules	**Ports**	**Description**	**Passed**	**Not Passed**	**Comments**
Digital out		**Set ports output from LOW to HIGH**			
-	1	Output should change states			
-	2	-			
-	3	-			
-	4	-			
Digital in		**Register states from digital out**			
-	1	Input should change states			
-	2	-			
-	3	-			
-	4	-			
Analog out		**Set voltage to 0V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$			
-	2	-			
-	3	-			
-	4	-			
-		**Set voltage to 2.5V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$			
-	2	-			
-	3	-			
-	4	-			
-		**Set voltage to 5V**			
-	1	Multimeter reads the voltage set within $\pm 0.05V$			
-	2	-			
-	3	-			
-	4	-			
Analog in		**Set up the analog in test-rig with 5V supply**			
-	1	Should Register 1V within $\pm 0.05V$			
-	2	Should Register 2V within $\pm 0.05V$			
-	3	Should Register 3V within $\pm 0.05V$			
-	4	Should Register 4V within $\pm 0.05V$			

3 Approval

By signing on this page, the individual listed has approved this test plan.

Name:

Date:

Bibliography

[1] S. Iqbal and A. Washim, “Programmable Logic Controllers (PLCs): Workhorse of Industrial Automation,” *IEEEP*, vol. 68-69, pp. 27–31, Apr. 2010. [Online]. Available: https://www.academia.edu/7063557/Programmable_Logic_Controllers_PLCs_Workhorse_of_Industrial_Automation

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7123563

[3] L. De Bernardini, “Industry 4.0 or Industrial Internet of Things - What’s Your Preference?” Aug. 2015. [Online]. Available: http://www.automationworld.com/industry-40-or-industrial-internet-things-whats-your-preference

[4] N. De Carne, “Putting the extra “I” in IoT,” Mar. 2015. [Online]. Available: https://www.linkedin.com/pulse/putting-extra-i-iot-nicola-de-carne

[5] S. Kinney, “APAC leading Industrial Internet of Things market growth,” Feb. 2016. [Online]. Available: http://www.rcrwireless.com/20160216/internet-of-things/industrial-internet-of-things-growth-led-by-apac-tag17-tag99

[6] G. Bora, S. Bora, S. Singh, and S. M. Arsalan, “OSI reference model: An overview,” *International Journal of Computer Trends and Technology (IJCTT)*, vol. 7, no. 4, pp. 214–218, 2014. [Online]. Available: http://ijcttjournal.org/Volume7/number-4/IJCTT-V7P151.pdf

[7] P. Seeberg, “Industry 4.0, OPC UA as a Bridge Between IT and Automation,” *Softing*, Jun. 2014. [Online]. Available: http://industrial.softing.com/uploads/softing_downloads/FA-Industrie-4_0-OPC-UA_EN.pdf

[8] K. Elleithy and T. Sobh, Eds., *Technological developments in networking, education and automation*. Dordrecht: Springer, 2010.

[9] S. Neil, “A Communication Gap in the Internet of Things,” Feb. 2015. [Online]. Available: http://www.automationworld.com/industrial-internet-things/communication-gap-internet-things

[10] B. Deborah, “Open Source Case for Business.” [Online]. Available: https://opensource.org/advocacy/case_for_business.php [Accessed: 22- May. 2016].

[11] M. Riedesser, “Controllino First Software Open-Source PLC.” [Online]. Available: http://controllino.biz/

[12] L. De Buck, “Industruino is an Arduino compatible industrial controller.” [Online]. Available: https://industruino.com/page/home

[13] E. Rossignon, "Build OPC UA applications in JavaScript and NodeJS." [Online]. Available: https://node-opcua.github.io/ [Accessed: 10- Apr. 2016].

[14] M. Purdy and L. Davarzani, "The Growth Game-Changer," 2015. [Online]. Available: http://skat.ihmc.us/rid=1P073HLQR-GZ8SNT-2Y1D/Accenture-Industrial-Internet-Things-Growth-Game-Changer.pdf

[15] "UaExpert, A Full Featured OPC UA Client," Feb. 2014. [Online]. Available: https://www.unified-automation.com/products/development-tools/uaexpert.html [Accessed: 1- Apr. 2016].

[16] L. A. Bryan and E. A. Bryan, *Programmable controllers: theory and implementation*, 2nd ed. Marietta, Ga: Industrial Text Comp. Publ, 1997.

[17] "Programmable Logic Controllers (PLCs) and PLC-based Programmable Automation Controllers (PACs)," Jun. 2015. [Online]. Available: http://www.arcweb.com/market-studies/pages/plcs-programmable-logic-controllers.aspx

[18] R. Davies, "Industry 4.0 Digitalisation for productivity and growth," Oct. 2015. [Online]. Available: http://www.europarl.europa.eu/RegData/etudes/BRIE/2015/568337/EPRS_BRI(2015)568337_EN.pdf

[19] "What is OPC?" [Online]. Available: https://opcfoundation.org/about/what-is-opc/

[20] "Mission Statement." [Online]. Available: https://opcfoundation.org/about/opc-foundation/mission-statement/

[21] "History." [Online]. Available: https://opcfoundation.org/about/opc-foundation/history/

[22] "Unified Architecture." [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/

[23] "Classic." [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-classic/

[24] S. Hoppe, "OPC Foundation Announces OPC UA Open Source Availability," Jun. 2016. [Online]. Available: https://opcfoundation.org/news/opc-foundation-news/opc-foundation-announces-opc-ua-open-source-availability/

[25] "Configuring ports for DCOM for use with the OPC Interface. NAT and Firewall considerations," Feb. 2015. [Online]. Available: https://techsupport.osisoft.com/Troubleshooting/KB/2973OSI8

[26] "OPC UA in Practice Users Workshop." Softing Industrial Automation GmbH, Jan. 2014. [Online]. Available: http://www.hitex.co.uk/fileadmin/uk-files/pdf/Automation%20seminars/online%20pdf%20downloads%20Coventry/05_OPC-UA-Workshop-Hitex-2014.pdf

[27] J. Fletcher, "Making a smooth transition from OPC Classic to OPC UA," Aug. 2016. [Online]. Available: http://www.controlengeurope.com/article/80950/Making-a-smooth-transition-from-OPC-Classic-to-OPC-UA.aspx

[28] "Welcome to the World of OPC." [Online]. Available: https://opcfoundation.org/

[29] J. Lange, F. Iwanitz, and T. J. Burke, Eds., *OPC: from Data Access to Unified Architecture*, 4th ed. Berlin: VDE-Verl, 2010.

[30] T. J. Burke, "OPC Unified Architecture Interoperability for Industrie 4.0 and the Internet of Things," OPC Foundation, Tech. Rep., 2015. [Online]. Available: https://opcfoundation.org/wp-content/uploads/2015/09/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN-v4.pdf

[31] "OPC Unified Architecture Part 2: Security Model," Nov. 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-2-security-model Release 1.03.

[32] "OPC Unified Architecture Part 4: Services Description," Jul. 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-4-services Release 1.03.

[33] "OPC Unified Architecture Part 7: Profiles," Nov. 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-7-profiles Release 1.03.

[34] G. Coley, "Beagleboard:BeagleBoneBlack," Feb. 2016. [Online]. Available: http://elinux.org/Beagleboard:BeagleBoneBlack [Accessed: 3- May. 2016].

[35] E. Upton, "Raspberry Pi 3 on sale now at $35," Feb. 2016. [Online]. Available: https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/

[36] H. Barragán, "The Untold History of Arduino," Jan. 2016. [Online]. Available: http://arduinohistory.github.io/

[37] D. Cuartielles, "Arduino FAQ – With David Cuartielles," May 2013. [Online]. Available: http://medea.mah.se/2013/04/arduino-faq/

[38] "Arduino Leonardo." [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardLeonardo

[39] F. Leens, "Introduction to I²C and SPI protocols." [Online]. Available: http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/

[40] "Raspberry Pi FAQs - Frequently Asked Questions." [Online]. Available: https://www.raspberrypi.org/help/faqs/ [Accessed: 25- Feb. 2016].

[41] "DS1307 64 x 8, Serial, I C Real-Time Clock," 2015. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS1307.pdf

[42] T. King, "Arduino Pin Current Limitations," Jan. 2016. [Online]. Available: http://arduino-info.wikispaces.com/ArduinoPinCurrent [Accessed: 20- Apr. 2016].

[43] "Arduino - AnalogRead." [Online]. Available: https://www.arduino.cc/en/Reference/AnalogRead

[44] K. Shirriff, "Secrets of Arduino PWM," Oct. 2016. [Online]. Available: http://www.righto.com/2009/07/secrets-of-arduino-pwm.html

[45] D. Yates, "Arduino Project 5: Digital audio player," 2013. [Online]. Available: http://apcmag.com/arduino-project-5-digital-audio-player.htm/

[46] "MCP4725 12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6," Oct. 2007. [Online]. Available: https://www.sparkfun.com/datasheets/BreakoutBoards/MCP4725.pdf

[47] L. Fried, “Adding a Real Time Clock to Raspberry Pi,” May 2015. [Online]. Available: https://learn.adafruit.com/adding-a-real-time-clock-to-raspberry-pi [Accessed: 5- Apr. 2016].

[48] I. Long, “CronHowto,” Apr. 2015. [Online]. Available: https://help.ubuntu.com/community/CronHowto [Accessed: 28- Apr. 2016].

[49] C. Robbins, “forever A simple CLI tool for ensuring that a given node script runs continuously (i.e. forever),” Jul. 2015. [Online]. Available: https://www.npmjs.com/package/forever [Accessed: 3- May. 2016].

[50] I. Z. Schlueter, “npm.” [Online]. Available: https://www.npmjs.com/

[51] E. Rossignon, “node-opcua an implementation of a OPC UA stack fully written in javascript and nodejs,” Apr. 2016. [Online]. Available: https://github.com/node-opcua/node-opcua [Accessed: 1- May. 2016].

[52] D. R. Baquero, “node-os NodeJS Core Module Extended,” Apr. 2016. [Online]. Available: https://github.com/DiegoRBaquero/node-os [Accessed: 20- Apr. 2016].

[53] Y. Korevec, “node-i2c - Node.js native bindings for i2c-dev. Plays well with Raspberry Pi and Beaglebone,” Aug. 2015. [Online]. Available: https://github.com/kelly/node-i2c [Accessed: 28- Apr. 2016].

[54] B. Cooke, “onoff - GPIO access and interrupt detection with JavaScript,” Mar. 2016. [Online]. Available: https://github.com/fivdi/onoff [Accessed: 18- Apr. 2016].

[55] “Arduino - FAQ.” [Online]. Available: https://www.arduino.cc/en/Main/FAQ

[56] L. Fried, “Adafruit MCP4725 12-bit I2c DAC,” Jan. 2016. [Online]. Available: https://github.com/adafruit/Adafruit_MCP4725 [Accessed: 24- Feb. 2016].

[57] J. Aro, “OPC UA Performance Evaluation,” Oct. 2015. [Online]. Available: https://www.youtube.com/watch?v=op8GqpH22Uw

[58] L. Clay, “SD cards,” Mar. 2016. [Online]. Available: https://www.raspberrypi.org/documentation/installation/sd-cards.md [Accessed: 1- Apr. 2016].