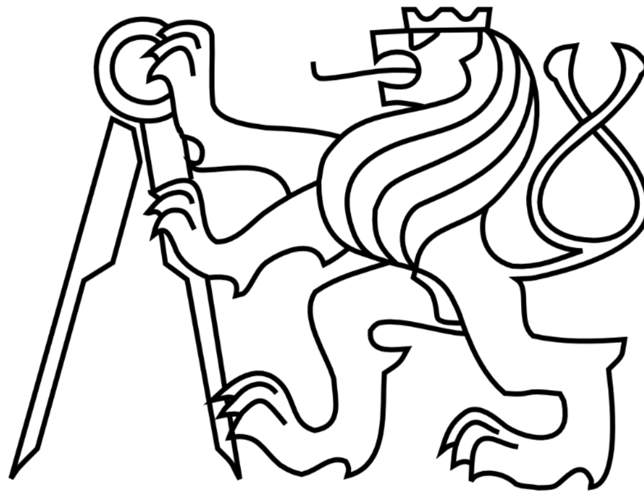
Czech Technical University in Prague

FACULTY OF ELECTRICAL ENGINEERING

DEPARTMENT OF TELECOMMUNICATIONS



Diploma Thesis

Integrace IO-Link OPC UA do zařízení Siemens SIMATIC
IO-Link OPC UA Integration for Siemens SIMATIC

Author: Rustambek Bekmukhamedov
Supervisor: Ing. Zbyněk Kocur, Ph.D.

Prague 2020

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Bekmukhamedov Rustambek** Personal ID number: **427554**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Microelectronics**

Study program: **Electronics and Communications**

Specialisation: **Electronics**

II. Master’s thesis details

Master’s thesis title in English:

IO-Link OPC UA Integration for Siemens SIMATIC

Master’s thesis title in Czech:

Integrace IO-Link OPC UA do zařízení Siemens SIMATIC

Guidelines:

1. The goal of the diploma work is the integration of IO-Link OPC UA to the SIEMENS SIMATIC system. The support of OPC UA for IO-Link is an advancement towards Industry 4.0.
2. The work begins with familiarization and original research on the OPC UA protocol and its applications for IO-Link. Search and analyze available solutions of OPC UA and IO-Link.
3. It continues with the proposition of the feature integration and its realization to the firmware.
4. Finally, verify the implementation by a demo application on the real hardware platform.

Bibliography / sources:

[1] IO-Link Community and OPC Foundation: OPC Unified Architecture for IO-Link, Companion Specification, Release V1.0
https://io-link.com/share/Downloads/OPC_UA/OPC-UA_for_IO-Link_10212_V10_Dec18.pd
f
[on-line] [2] 6GT2002-0JE50: https://mall.industry.siemens.com/mall/en/WW/Catalog/Product/6GT2002-0JE50 [on-line]
https://automationinside.com/article/new-device-series-paves-the-way-for-high-fr
equency-rfid-cloud-connection
[on-line] [3] 6ES7647-0AA00-1YA2: https://support.industry.siemens.com/cs/pd/815412?pdti=pi&dl=en&lc=en-WW [on-line]

Name and workplace of master’s thesis supervisor:

Ing. Zbyněk Kocur, Ph.D., Department of Telecommunications Engineering, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **05.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Zbyněk Kocur, Ph.D. Supervisor’s signature	prof. Ing. Pavel Hazdra, CSc. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt — Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Table of Contents

1. Introduction **8**
2. Theoretical overview **9**
2.1 IO-Link 9
2.1.1 System Overview 10
2.1.2 Interface 11
2.1.3 Communication startup 13
2.1.4 Device Profiles 14
2.1.5 IO Device Description 14
2.1.6 Configuration tool for IO-Link 15
2.1.7 Configuring IO-Link in automation system 16
3. OPC UA **16**
3.1 Introduction 16
3.2 Specification 18
3.3 Foundation 19
3.3.1 Transport 20
3.3.2 Data Model 22
3.4 Client-Server 23
3.4.1 Client 24
3.4.2 Server 25
3.5 Address Space 26
3.5.1 Node Model 27
3.5.2 Node Classes 28
3.5.3 NodeId 31
4. Task motivation and solution **33**
4.1 Motivation 33
4.2 Solution proposal 33
4.2.1 Task steps 35
5. OPC UA SDK **35**
5.1 IO-Link Information model 35
5.1.1 Model Overview 36
5.1.2 SiOME 37
5.1.3 Creating IOLinkDeviceType in SiOME 41
5.1.4 Generating code files using UaModeler 43
5.1.5 Generating dynamic address space binaries 45

5.2 Building application in SDK environment ... 45
5.2.1 Integrating into CMAKE file ... 46
5.2.2 Adding Value Store ... 46
5.2.3 Adding Method ... 48
5.3 Cross-compiling for Raspberry Pi Zero ... 54
5.3.1 Creating target configuration file ... 55
5.3.2 Creating toolchain CMake file ... 55
5.3.3 Running build script and uploading binaries to the SDK ... 55
5.3.4 Connecting to server using UaExpert and verifying functionality ... 55
6. Conclusion ... 61
7. References ... 62

Declaration

I hereby declare that this thesis is the result of my own work and that I have clearly stated all information sources used in the thesis according to "Methodological Instructions of Ethical Principle in the Preparation of University Thesis".

In Prague, 14.08.2020 Signature .....………………….

Acknowledgments

I would like to express my gratitude and appreciation for my supervisor Ing. Zbyněk Kocur, Ph.D for his support throughout my Diploma thesis, advice and help from my colleagues Ing. Miloš Fenyk and Ing. Martin Huncovsky at Siemens. I would also like to thank my family for their continuous support during my studies.

Abstract

This diploma thesis is dedicated to the implementation of OPC UA support for IO Link in the SIEMENS SIMATIC System. In this thesis I will present theoretical background on the specifications of the OPC UA and IO-Link communication protocols and explore the features of OPC UA SDK for bridging the functionality between OPC UA and IO-Link. The project continues with the definition and proposition of the design of integration with the firmware. A use case of this integration will be implemented for Raspberry Pi board running on Linux, which acts as an IO-Link Master. The workings of the implementation are verified by a demonstration application on the Raspberry Pi board.

Keywords

OPC UA, IO-Link, Raspberry PI Zero

List of Abbreviations

OPC UA: Open Platform Communications Unified Architecture
PLC: Programmable Logic Controller
HMI: Human-Machine Interface
IODD: Electronic device description of devices (IO Device Description)
GSD: Generic Station Description
DI: Digital Input
DQ: Digital Output
ERP: Enterprise Resource Planning
HTTP: Hypertext Transfer Protocol
IP: Internet Protocol
ISDU: Indexed Service Data Unit
MES: Manufacturing Execution System
PMS: Production Management System
SCADA: Supervisory Control and Data Acquisition
TCP: Transmission Control Protocol
XML: Extensible Markup Language
IoT: Internet of Things
SoC: System on Chip

1. Introduction

The developments of steam powered engines brought upon us the first industrial revolution, where engineers built large scale machinery capable of undertaking the task of tens and hundreds of men. This powered everything from textile to agriculture. With steam power, agrarian societies gave way to urbanization. Furthermore, as science evolved its fruits no longer became limited to the laboratory. Scientific principles were brought right into factories. The invention of gasoline engines, airplanes and chemical fertilizers and the introduction of assembly lines in factories gave way to the second industrial revolution. The advancements of semiconductor technology and digital electronics has resulted in the third revolution – the digital revolution, the products of which has become part of our everyday life. We are now witnessing the rise of the fourth industrial revolution, as self-learning algorithms, cloud computing and IoT systems are more integrated into our society.

The ongoing initiative which plans to transform the manufacturing industry is called Industry 4.0. It has been defined as the current trend of automation and data exchange in manufacturing technologies, including cyber-physical systems, the Internet of Things, cloud computing and artificial intelligence in creating the “smart factory”.

Siemens is the largest industrial manufacturing company in Europe focusing on automatization and digitization of factories, while pushing towards the development of Industry 4.0 and building the necessary products and infrastructure. The small subset task, which will advance the future capabilities of SIEMENS SIMATIC products, is the combination of OPC UA with IO-Link. Both OPC UA and IO-Link are industry standard communication protocols. While IO-Link is a point-to-point, fast and simple protocol [1], OPC UA is a routed, packet based and secure protocol with internetwork and over the internet communication capabilities [2]. The mapping of IO-Link data to OPC UA data model would have many beneficial use cases such as parametrization of IO-Link devices, collection of diagnostic data by OPC UA Client over OPC UA and many more [2]. Different deploy options of OPC UA with IO-Link can be seen on Figure 1. The OPC UA Server can directly be deployed on an IO-Link Master or a PLC connected to the IO-Link Master or another platform like a PC [2]. The OPC UA Client can directly be connected to the OPC UA Server running on the IO-Link Master, it can be connected to the PLC running the OPC UA Server, or the PLC can forward the traffic from an OPC UA Client on top of the PLC to the OPC UA Server running on the IO-Link Master beneath the PLC [2].

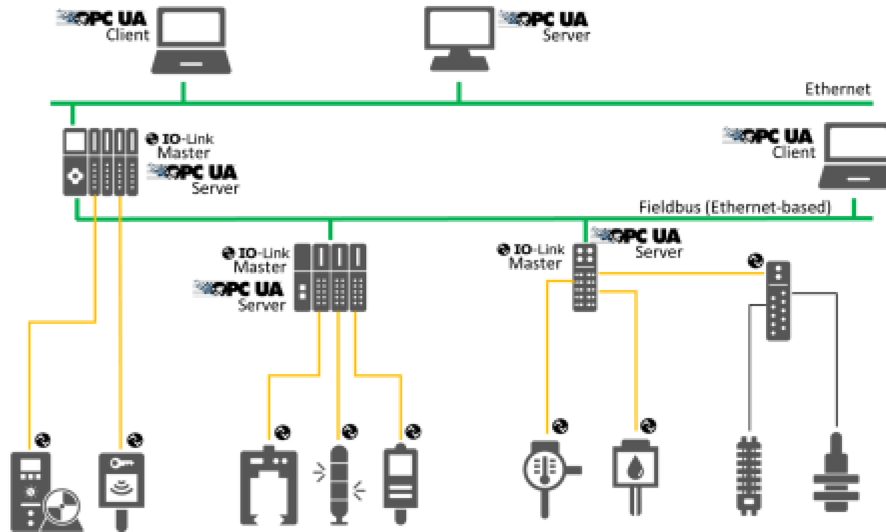


Figure 1: System Architecture of IO-Link and OPC UA [2].

2. Theoretical overview

2.1 IO-Link

IO-Link is an internationally standardized (IEC 61131-9) industrial communications networking protocol. It is described as short distance, bi-directional, point to point, serial communications protocol, used for connecting digital sensors and actuators to either a type of industrial fieldbus or a type of industrial Ethernet (Figure 2) [1]. It is based on the long established 3-wire sensor and actuator connection without additional requirements regarding the cable material. So, IO-Link is no fieldbus but the further development of the existing, tried-and-tested connection technology for sensors and actuators. It is considered as a powerful standard in the automatization industry as it is fieldbus independent and can be integrated into all fieldbus systems worldwide [1]. IO-Link relies on standards such as M12, M8 or M5 connectors and three-wire cables, which contributes to uniform interface for sensors and actuators irrespective of their complexity (switching, measuring, multi-channel binary, mixed signal, etc.). It allows three types of data to be exchanged – Process data, service data, and events [1].

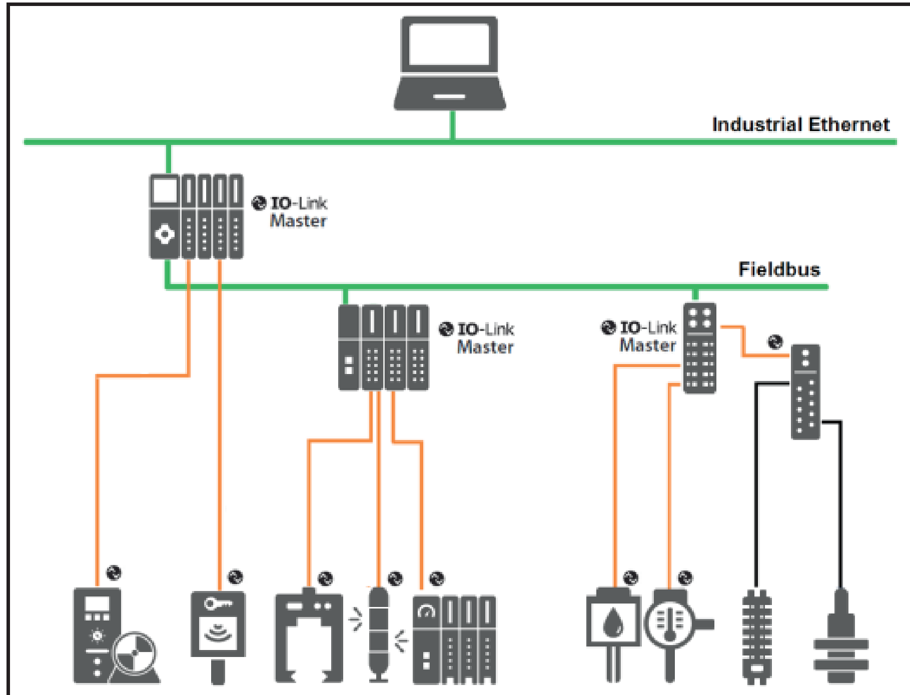


Figure 2: Example of system architecture with IO-Link [1].

2.1.1 System Overview

As it can be seen from the general example of a system architecture with IO-Link (Fig.1), we differentiate the following basic components: IO-Link Master, IO-Link device (e.g., sensors, RFID readers, valves, I/O modules), unshielded 3- or 5-conductor standard cables and devices for configuring and assigning parameters of IO-Link. The IO-Link master establishes the connection between the IO-Link devices and the automation system [1]. As a component of an I/O system, the IO-Link master is installed either in the control cabinet or as remote I/O, with enclosure rating of IP65/67, directly in the field [1]. The IO-Link master communicates over various fieldbuses or product-specific backplane buses [1]. An IO-Link master can have several IO-Link ports (channels). An IO-Link device can be connected to each port (point-to-point communication). Hence, IO-Link is a point-to-point communication and not a fieldbus (Fig.2). The engineering of the IO-Link system is performed in parallel with the engineering of the overall automation system and can be embedded in and meshed with this engineering [1].

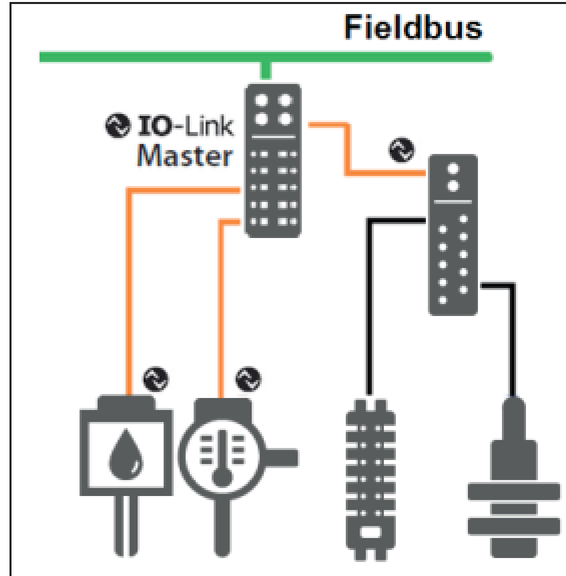


Figure 3: IO-Link point to point connection diagram [1].

2.1.2 Interface

As mentioned before, IO-Link is a serial, bi-directional point-to-point connection for signal transmission and energy supply under any networks, fieldbuses, or backplane buses. For the connection technology in IP65/67, one possibility that has been defined is an M12 plug connector, in which sensors usually have a 4-pin plug and actuators a 5-pin plug. IO-Link masters generally have a 5-pin M12 socket. The pin assignment is specified according to IEC 60974-5-2 as follows [1]:

- Pin 1: 24 V
- Pin 3: 0 V
- Pin 4: Switching and communication line (C/Q)

These three pins not only provide IO-Link communication, but also supply the device with at least 200 mA (Fig.3).

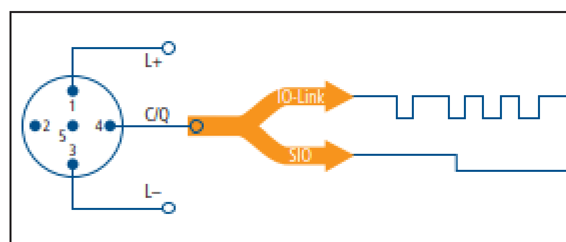


Figure 4: Pin assignment of IO-Link device [1].

There are two types of ports in IP65/67 for the IO-Link Master: Port Class A (Type A) and Port Class B (Type B). They differ by how they utilize the ports 2 and 5 in Fig.3 [1]. In Type A, the functions of pins 2 and 5 are not specified. The manufacturer defines these functions. Pin 2 is usually assigned with an additional digital channel used as Digital Input or Output. Type B provides additional supply voltage and is suitable for the connection of devices that have an increased power demand. In this case, pins 2 and 5 are used to provide additional (galvanically isolated) supply voltage. A 5-conductor standard cable is required in order to use this additional supply voltage [1].

The device is connected to the master via unshielded 3 or 5-lead standard cables with a length extending up to 20 m with cross-section >= 0.34 mm^2. Shielding is not necessary. Likewise, no specific guidelines have to be followed when laying the cables.

The IO-Link ports are capable of operating at different modes: “IO-Link”, “DI”, “DQ” and “Deactivated” modes. “IO-Link” is used for IO-Link communication [1]. In “DI” mode the port behaves like a digital input and in “DQ” – digital output. “Deactivated” ports as the name suggests are inactive [1][3].

The version 1.1 of IO-Link specification presents 3 transmission rates [1]:

- COM 1 = 4.8 kbaud
- COM 2 = 38.4 kbaud
- COM 3 = 230.4 kbaud

IO devices support inly one of the defined data transmission rates, whereas IO master supports all data transmission rates and adapts itself automatically to the data transmission rate supported by the device [1][3].

The response time of the IO-Link system provides information about the frequency and speed of the data transmission between the device and master. The response time depends on various factors. The device description file IODD of the device contains a value for the minimum cycle time of the device. This value indicates the time intervals at which the master may address the device. The value has a large influence on the response time. In addition, the master has an internal processing time that is included in the calculation of the response time [1][3]. Devices with different minimum cycle times can be configured on one master. The response time differs accordingly for these devices. That is, the response time of the different devices on a master can differ significantly. When configuring the master, you can specify a fixed cycle time in addition to the device-specific minimum cycle time stored in the IODD [1][3]. The master then addresses the device based on this specification. The typical response time for a device

therefore results from the effective cycle time of the device and the typical internal processing time of the master [1][3].

IO-Link is a very robust communication system. This communication system operates with a 24 V level. If transmissions fail, the frame is repeated two more times. After second failed attempt to send or receive, the IO-Link master recognizes a communication failure and signal this to the higher-level controller.

There are several types of data that are exchanged through IO-Link, some are exchanged in cyclic, while others in acyclic manner [1]. Cyclic data is exchanged regularly and periodically, such as process data and value status. Acyclic data is exchanged when needed and occur only when certain events trigger them, such as device data and events [1].

The process data of the devices are transmitted cyclically in a data frame in which the size of the process data is specified by the device. Depending on the device, 0 to 32 bytes of process data are possible (for each input and output) [1]. The consistency width of the trans-mission is not fixed and is thus dependent on the master. Each port has a value status (e.g, PortQualifier). The value status indicates whether the process data are valid or invalid. The value status is transmitted cyclically with the process data [1].

Device data can be parameters, identification data, and diagnostic information. They are exchanged acyclically and at the request of the IO-Link master. Device data can be written to the device (Write) and also read from the device (Read).

When an event occurs, the device signals the presence of the event to the master. The master then reads out the event. Events can be error messages (e.g., short-circuit) and warnings/maintenance data (e.g., soiling, overheating) [1][3].

2.1.3 Communication startup

If the port of the master is set to IO-Link mode, the IO-Link master attempts to communicate with the connected IO-Link device. To do so, the IO-Link master sends a defined signal (wake up pulse) and waits for the IO-Link device to reply. The IO-Link master initially attempts to communicate at the highest defined data transmission rate. If unsuccessful, the IO-Link master then attempts to communicate at the next lower data transmission rate[1][3]. The device always supports only one defined data trans-mission rate. If the master receives a reply, the communication begins. For this purpose, the master and device exchange the necessary communication parameters. If data management is activated, the parameters stored in the master are transferred to the device. Then, the cyclic exchange of the process data and value status begins [1][3]

2.1.4 Device Profiles

To standardize how the user program on the controller accesses the devices, device profiles are defined for IO-Link. The device profiles specify the data structure, data contents, and basic functionality. As a result, a uniform user view and an identical access by the program on the controller is achieved for a variety of different devices that conform to the same device profile. Current version defines three profiles [1]:

- Binary switching sensors
- Digital measuring sensors
- Uniform system behavior devices

The profiles for switching sensors are appropriate for simple switching applications such as presence detection.

The digital measurement profiles are designed for measuring sensors that can provide the values on the basis of a physical unit such as temperature or pressure. These profiles make it possible to integrate IO-Link sensors without special knowledge of the sensor used [1].

The device profile for uniform system behavior defines minimum device identification, diagnostics and event data [1][3]. IO-Link devices that support this profile offer a uniform minimum degree of integration of the system into the controller. This profile is the basis for all other device profiles [1][3].

2.1.5 IO Device Description

Each IO-Link Device has an IODD (IO Device Description). This is a device description file which contains information about the manufacturer, article number, functionality etc. This information can be easily read and processed by the user. Each device can be unambiguously identified via the IODD as well as via an internal device ID [2]. The structure of the IODD is the same for all devices of all manufacturers. The structure of the IODD is always represented in the same way by the IO-Link configuration tools of the master manufacturers. This ensures the same handling of all IO-Link devices irrespective of the manufacturer. For devices that support both V1.0 and V1.1 functionality, two different IODD versions are available [2].

2.1.6 Configuration tool for IO-Link

In order to configure the entire IO-Link system, configuration tools are required. The IO-Link configuration tools of the master manufacturers are able to read in IODDs (Figure 5). The main tasks that the configuration tool must carry on include [1]:

- Assignment of the devices to the ports of the master
- Address assignment (I/O addresses of the process data) to the ports within the address area of the master
- Parameter assignment of the IO-Link devices

Figure 5: Configuration tool with IODD of a device and the device information it contains [1].

2.1.7 Configuring IO-Link in automation system

The configuration of IO-Link can be simplified in two steps. In the first step, the IO-Link master is connected to the automation system and configured. In the second step, the IO-Link device parameters are set. In the configuration of the automation system or fieldbus, the IO-Link system is represented by the IO-Link master and integrated using the appropriate device description (e.g., GSD file for PROFINET) [1][3]. The IO-Link master itself can be a fieldbus node or a module of a modular IO system that is connected to the fieldbus. In both cases, the number of ports, the address range, and the module properties are described in the device description of the IO-Link master [1][3].

Figure 6 shows a PROFINET configuration into which PROFINET devices with IO-Link masters are integrated.

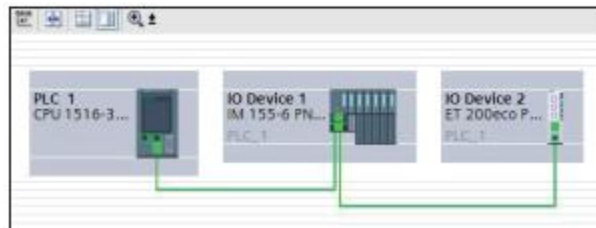


Figure 6: Configuring SIEMENS PLC PROFINET connection with IO-Link masters connected through an Interface Module [1].

In the device view of the hardware configuration, the input and output address ranges for the exchange of cyclic data (process values) of IO-Link are specified [1]. Furthermore, it is possible to specify in the module properties of the IO-Link master how the port configuration should be set. In the process, it is possible to choose whether to work with or without an additional IO-Link configuration tool [1].

3. OPC UA

3.1 Introduction

OPC UA is an open and royalty free set of standards designed as a universal communication protocol. While there are numerous communication solutions available, OPC UA differentiates itself with the following advantages [2]:

- A state of art security model
- A fault tolerant communication protocol

- An information modelling framework that allows application developers to represent their data in a way that makes sense to them.

OPC UA can be mapped onto a variety of communication protocols and data can be encoded in various ways to trade off portability and efficiency. As an open standard, OPC UA communicates on standard internet technologies, like TCP/IP, HTTP, Web Sockets [2]. OPC UA is designed to provide robustness of published data. A major feature of all OPC servers is the ability to publish data and Event Notifications. OPC UA provides mechanisms for Clients to quickly detect and recover from communication failures associated with these transfers without having to wait for long timeouts provided by the underlying protocols.[7]

OPC UA has a broad scope which delivers for economies of scale for application developers. This means that a larger number of high-quality applications at a reasonable cost are available. When combined with semantic models such as OPC UA for IO-Link or any other communication standard, OPC UA makes it easier for end users to access data via generic commercial applications.

The OPC UA model is scalable from small devices to ERP systems. OPC UA Servers process information locally and then provides that data in a consistent format to any application requesting data - ERP, MES, PMS, Maintenance Systems, HMI, Smartphone or a standard browser [2].

As an extensible standard, OPC UA provides a set of Services and a basic information model framework [2]. This framework provides an easy manner for creating and exposing vendor defined information in a standard way. More importantly all OPC UA Clients are expected to be able to discover and use vendor-defined information [2]. This means OPC UA users can benefit from the economies of scale that come with generic visualization and historian applications.

OPC UA Clients can be any consumer of data from another device on the network to browser based thin clients and ERP systems. The full scope of OPC UA applications is shown in Figure 7.

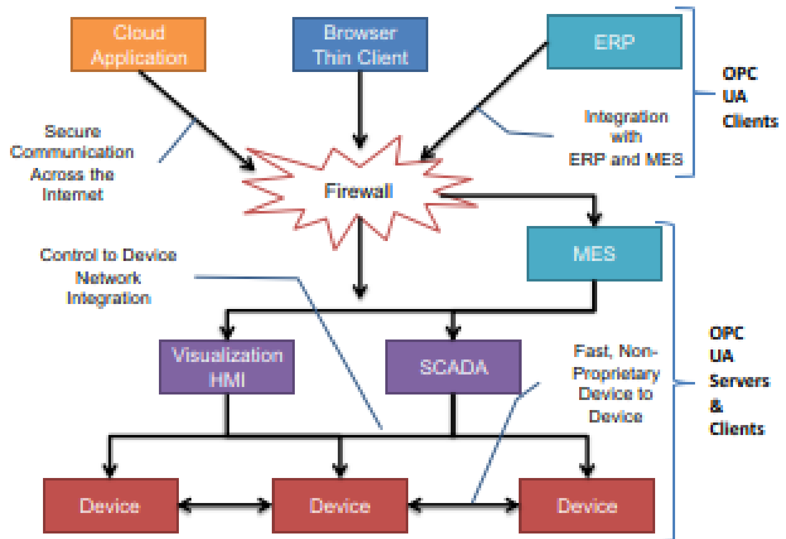


Figure 7: Scope of OPC UA within Enterprise [2].

OPC UA provides a robust and reliable communication infrastructure having mechanisms for handling lost messages, failover, heartbeat, etc [2]. With its binary encoded data, it offers a high -performing data exchange solution [2]. Security is built into OPC UA as security requirements become more and more important especially since environments are connected to the office network or the internet and attackers are starting to focus on automation systems.

3.2 Specification

The whole OPC UA specification is organized into multiple parts as shown in Figure 8.

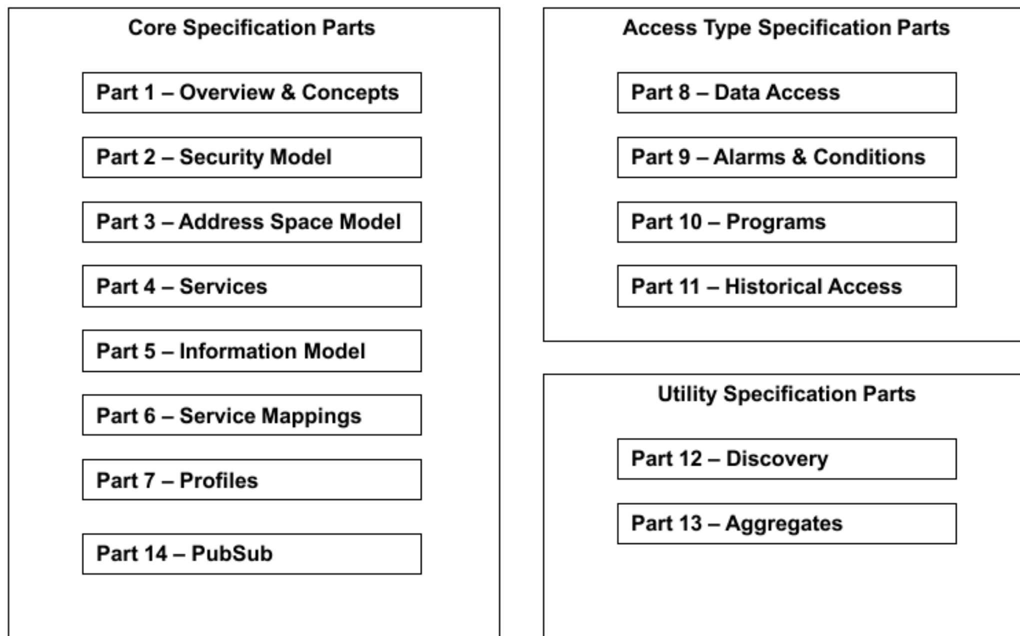


Figure 8: OPC UA specification in parts [7].

The base of OPC UA is defined by the core specification parts. These core capabilities define the structure of the OPC Address Space and the Services that operate on it. Access Type parts describe how core capabilities are applied to specific types of access. We don't require to research of each of these documents in detail. The essential parts that concern us are Part 3 and 4 which are important for the design and development of OPC UA applications.

The Address Space Model in UA Part 3 specifies the building blocks to expose instance and type information and thus the OPC UA meta model used to describe and expose information models and to build an OPC UA server address space. The abstract UA Services defined in UA Part 4 represent the possible interactions between UA client and UA server applications. The client uses the Services to find and access information in the Address Space. The Services are abstract because they are defining the information to be exchanged between UA applications but not the concrete representation on the wire and also not the concrete representation in an API used by the applications [7].

3.3 Foundation

The OPC UA builds on different layers shown in Figure 8.

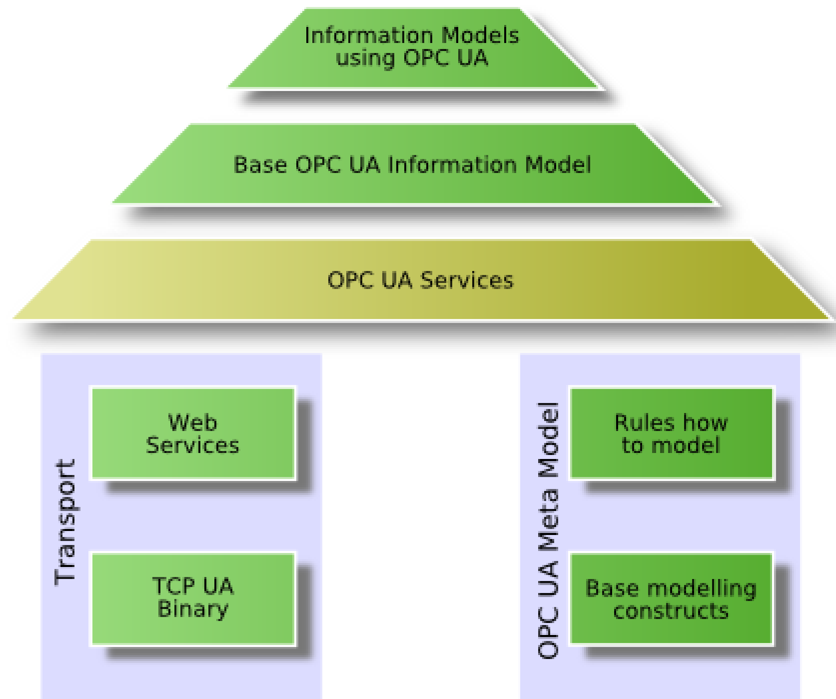


Figure 9: OPC UA component Layers [4].

The fundamental components of OPC Unified Architecture are transport mechanisms and data modeling.

3.3.1 Transport

The transport provides different mechanisms optimized for different use cases. The first version of OPC UA utilizes an optimized binary TCP protocol for high performance intranet communication as well as a mapping to accepted internet standards like Web Services, XML, and HTTP for firewall-friendly internet communication [7] as shown in Figure 10, although future version are deprecating XML and HTTP and opting for HTTPS instead. There are also mixed variant "protocol bindings" or hybrid variants, which combine two. All three variants can be used in parallel. An application programmer will only observe this due to the different URL he or she has to pass: opc.tcp://server for the binary protocol and http://server for WebService. Apart from that OPC UA works completely transparent with respect to the API [7]. Thus, application developers can switch between protocol bindings without having to adapt or reimplement.

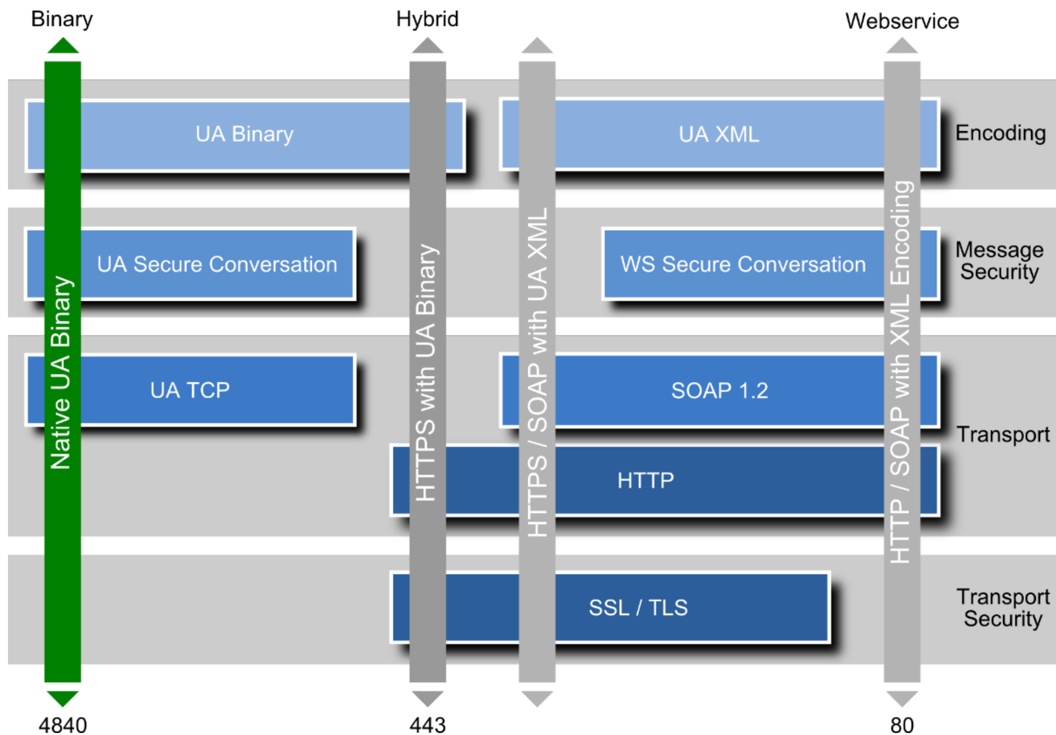


Figure 10: OPC UA Communication protocols [8].

The supported protocols and their individual properties can be summarized as follows:

1. Binary protocol (UA binary):
   - Mandatory
   - best performance, smallest overhead [7]
   - takes minimum resources (no XML Parser, SOAP and HTTP required which is important for embedded devices) [7]
   - best possible interoperability (binary is explicitly specified and allows less choices during implementation than XML does) [7]
   - only one single TCP port (4840) is used for communication and can easily be used for tunneling or enabled through a firewall [7]

2. Webservice (XML-SOAP)
   - optional, additional
   - extensively supported by available tools, can e.g. easily be used out of Java or .NET environments [7]
   - firewall-friendly; port 443 (https) will usually work without additional configuration [7]

3. Hybrid (UA-Binary via HTTPS)
   - optional
   - less overhead than XML-SOAP [7]
   - combines the advantages of both protocols: binary encoded payload in a HTTPS frame [7]
   - firewall-friendly; port 443 (https) will usually work without additional configuration [7]

The mapping of the UA Services to messages, the security mechanisms applied to the messages, and the concrete wire transport of the messages are defined in UA Part 6. [7] Figure 11 shows the layered communication architecture of OPC UA, and how the service Part 4 and communication mapping Part 6 related in that regard.

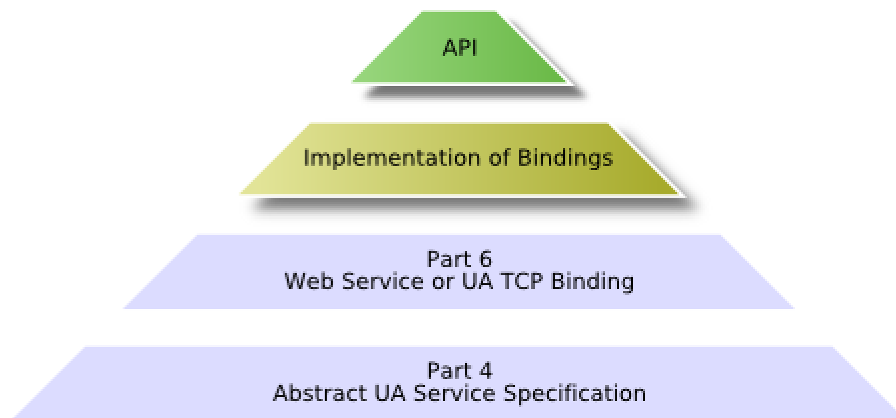


Figure 11: Layered OPC UA communication architecture [4].

As OPC UA architecture is a Service Orientated Architecture (SOA) and is based on different logical levels. All of the Base Services defined by OPC shown in Figure 12 are abstract method descriptions which are protocol independent and provide the basis for the whole OPC UA functionality.[4]

3.3.2 Data Model

The data modeling defines the rules and base building blocks necessary to expose an information model with OPC UA. It defines also the entry points into the address space and base types used to build a type hierarchy. This base can be extended by information models building on top of the abstract modeling concepts. In addition, it defines some enhanced concepts like describing state machines used in different information models.[7]

The UA Services serve as interface between servers and client, where servers act as supplier of an information model and clients - as consumers of that information model. The Services are defined in an abstract manner. They are using the transport mechanisms to exchange the data between client and server.[7] Figure 12 shows this layered relationship between the service and information models.

The benefit of OPC UA is that it simplifies the access to smaller elements of data without the need of understanding the complex system by whole. Figure12 shows the different layers of information models defined by OPC, by other organizations, or by vendors.[4]

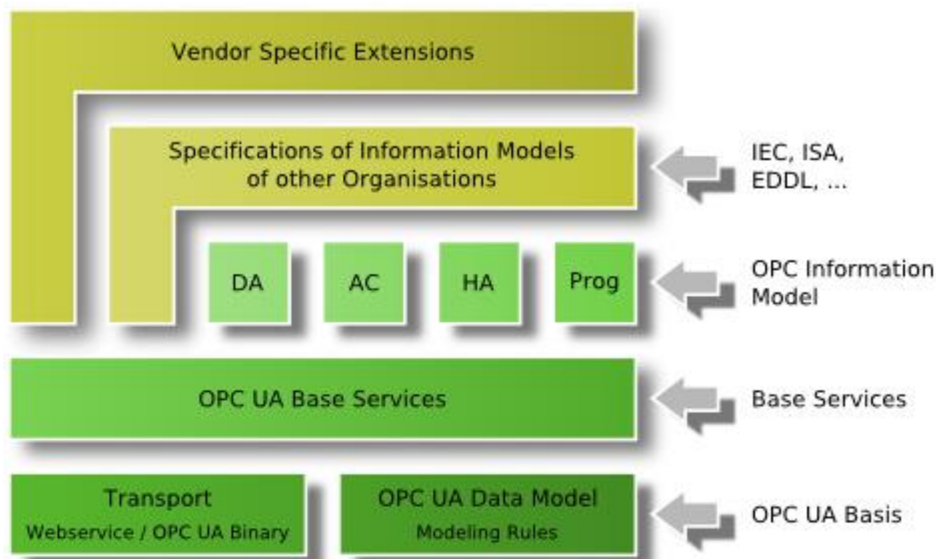


Figure 12: OPC UA Layers of information models.[4].

Information models for the domain of process information are defined by OPC UA on top of the base services and specifications (Figure 12). Data Access (DA) defines automation-data-specific extensions such as the modeling of analog or discrete data and how to expose quality of service [4]. All other DA features are already covered by the base. Alarm & Conditions (AC) specifies an advanced model for process alarm management and condition monitoring. Historical Access (HA) defines the mechanisms to access historical data and historical events. Programs (Prog) specifies a mechanism to start, manipulate, and monitor the execution of programs.[4]

3.4 **Client-Server**

Usually OPC UA is used in the Client-Server communication model. In the Client Server model, as mentioned before, OPC UA defines sets of Services that Servers may provide, and individual Servers

specify to Clients what Service sets they support. Information is conveyed using OPC UA- defined and vendor-defined data types, and Servers define object models that Clients can dynamically discover. [7] Servers can provide access to both current and historical data, as well as Alarms and Events to notify Clients of important changes. In addition to the Client Server model, OPC UA defines a mechanism for Publishers to transfer the information to Subscribers using the PubSub model.[7]

3.4.1 Client

Figure 13 illustrates an example of general Client model, its interaction, some major elements and how they relate together. The Client application is represented as code running as the topmost layer of the Client on Figure 13. It contains specific functionality for the application and the mapping of this functionality to OPC UA by using an OPC UA Stack and an OPC UA SDK.

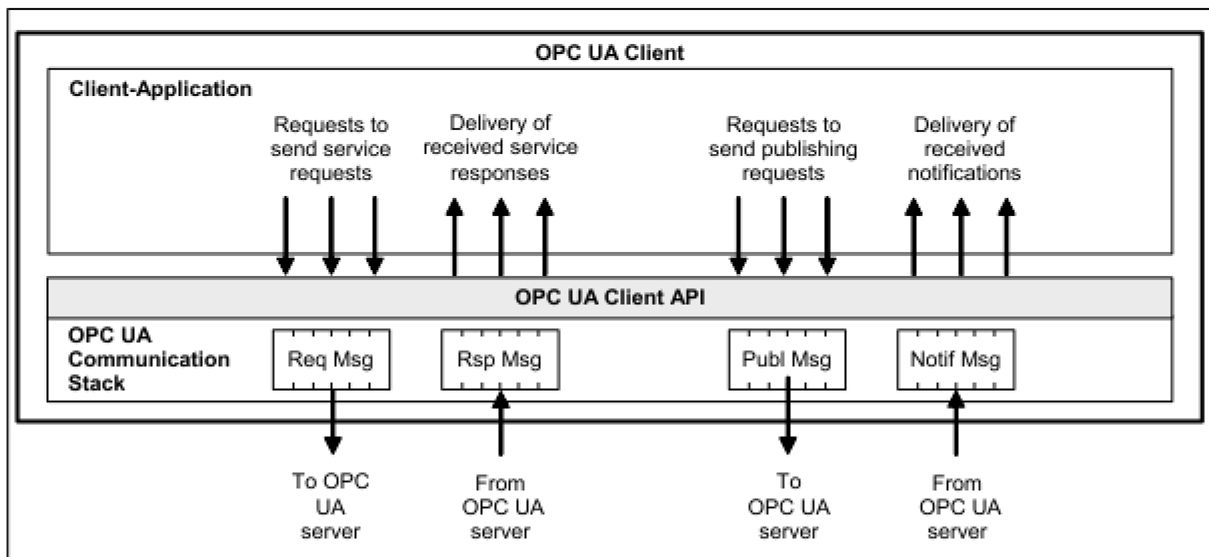


Figure 13: OPC UA Client architecture [7].

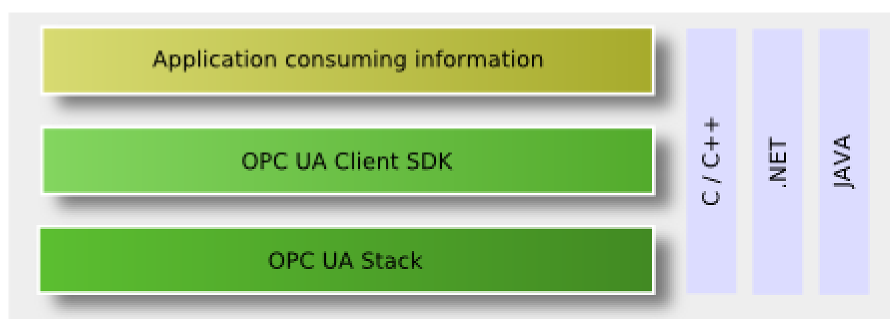


Figure 14: Software Layers of OPC UA Client [4].

3.4.2 Server

The following Figure 15 illustrates the OPC UA Server architecture model. As in the case of the Client application, the Server application is the code that implements the function of the Server. It has a similar software layer structure as seen on Figure 14, except the Application layer can be seen as providing information.

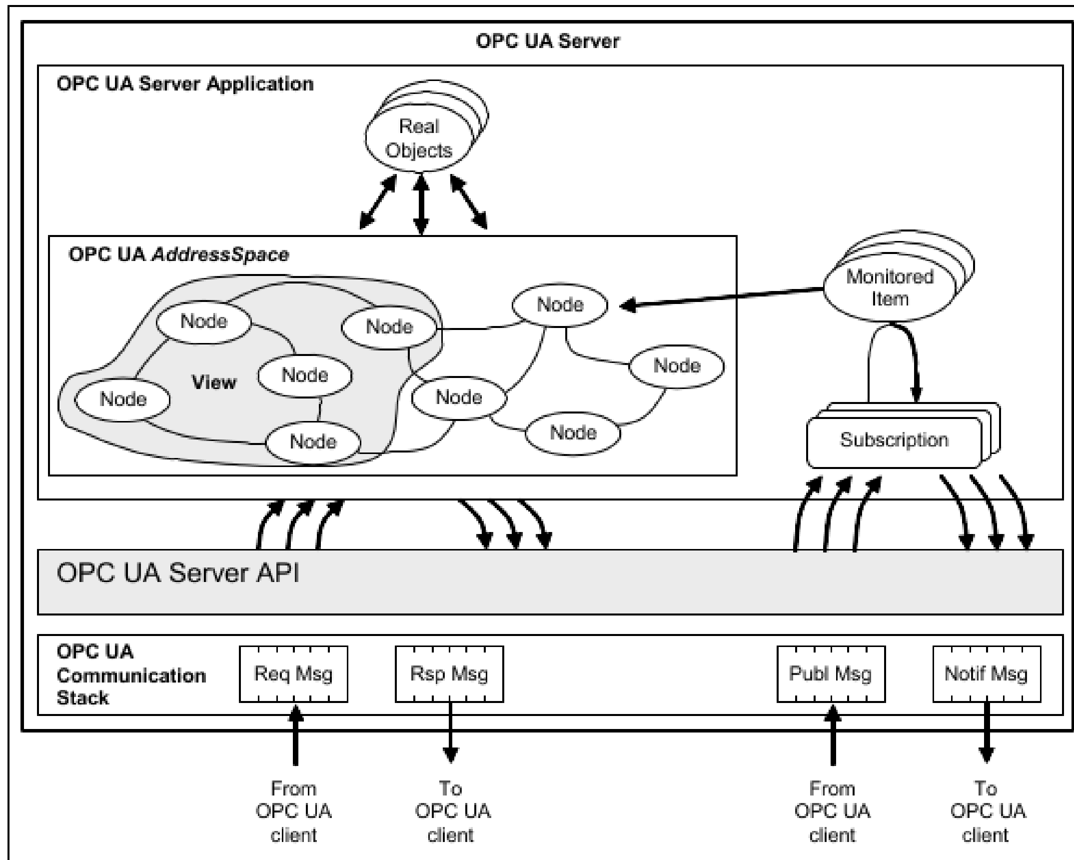


Figure 15: OPC UA Server architecture [7].

Real objects presented in Figure 15, are physical or software objects that are accessible by the Server application or that it maintains internally. Examples include a connected sensor device and diagnostics counters or current date and time.[7]

The Address Space represents the information accessible by clients and is modelled as a set of Nodes. Using OPC UA Services (interfaces and methods) client can read/write variable or call functions. Nodes in the Address Space are used to represent real objects, their definitions and their references to each other.[7]

The OPC UA Address Space supports information models. Information model, as mentioned, are a structural representation of real-world entities using object model similar to OOP. The Address Space represents these entities, its components and relation of components or relation of entities using Nodes and its constructs.[9]

There are two, so called "Subscription entities", MonitoredItems and Subscriptions in the Server.[7] MonitoredItems are entities in the Server created by the Client that monitor Address Space Nodes and their real-world counterparts. When they detect a data change or an event/alarm occurrence, they generate a Notification that is transferred to the Client by a Subscription.[7] A Subscription is an endpoint in the Server that publishes Notifications to Clients. Clients control the rate at which publishing occurs by sending Publish Messages.[7]

3.5 Address Space

An Address Space provides the entry point for Clients for browsing and accessing data, subscribing for event notifications or calling methods and is standard way for Servers to represent Objects. As it was mentioned, the Address Space is defined by Part 3 of the OPC UA specification.[9] Similar to OOP, The OPC UA Object Model has been designed to meet this objective. It defines Objects in terms of Variables and Methods.[7] It allows expressing the relationship between different Objects and Figure 16 illustrates the model.

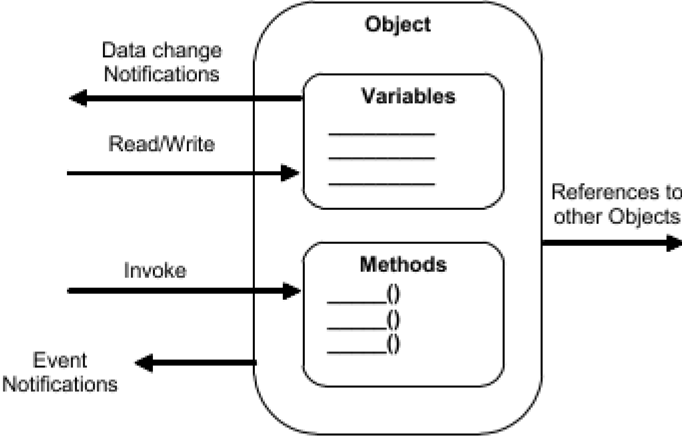


Figure 16: OPC UA Object Model [9].

The UA services are used to access the objects and their components like reading or writing a variable value, calling a method or receiving events from the object. The browse service can be used to explore relationships between objects and their components.

This is where we introduce the Node model and tie it with the Object model as elements of this model are represented in the Address Space as Nodes. There are various Node Classes that each represent different element of the Object Model and each Node belong to a particular class.[9]

3.5.1 Node Model

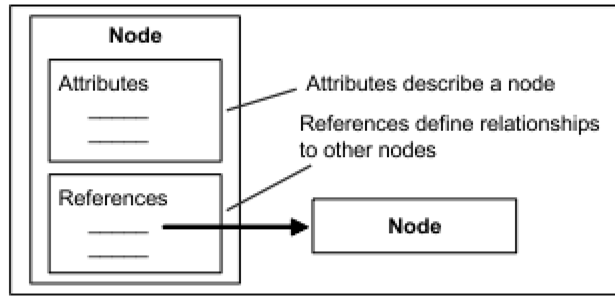


Figure 17: Address Space Node model [9].

As we can see in Figure 17, Nodes are described by attributes and interconnected by references.

3.5.1.1 Attributes

Attributes are elementary components of NodeClasses and are data elements that describe Nodes [7]. Clients can use the Read, Write, Query, and Subscription/MonitoredItem Services [10] to access Attribute values. Each attribute definition consists of an attribute id, a name, a description, a data type and a mandatory/optional indicator. The set of attributes defined for each node class cannot be extended by clients or servers. When a node is instantiated in the address space, the values of the mandatory node class Attributes must be provided.[9]

3.5.1.2 References

References, as the name suggests, are used to relate nodes to each other. They play an important role in structuring the data in the information model and enable browsing and querying services.[4] Similar to attributes, they are defined as fundamental components of nodes but unlike attributes, references are defined as instances of Reference Type nodes [4]. Reference Type nodes are visible in the address space and are defined using the Reference Type node class.[9] The node that contains the reference is referred to as the source node (Figure 18) and the node that is referenced is referred to as the target node. The combination of the source node, the Reference Type and the target node are used in OPC UA services to uniquely identify references [9]. Thus, each node can reference another node with the same Reference Type only once [9]. The target node of a reference may be in the same address space or in the address

space of another OPC UA server [4]. Target nodes located in other servers are identified in OPC UA services using a combination of the remote server name and the identifier assigned to the Node by the remote server. [4]

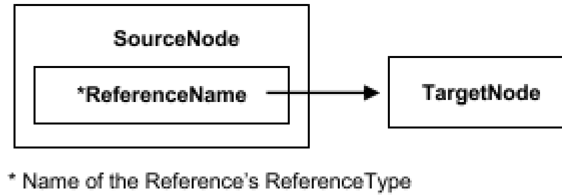


Figure 18: Reference Model [9].

3.5.2 Node Classes

Figure 19 illustrates eight node classes defined in OPC UA. Each node in the address space is an instance of one of these node classes. Clients and servers are not allowed to define additional node classes or extend the list of attributes of these node classes [4].

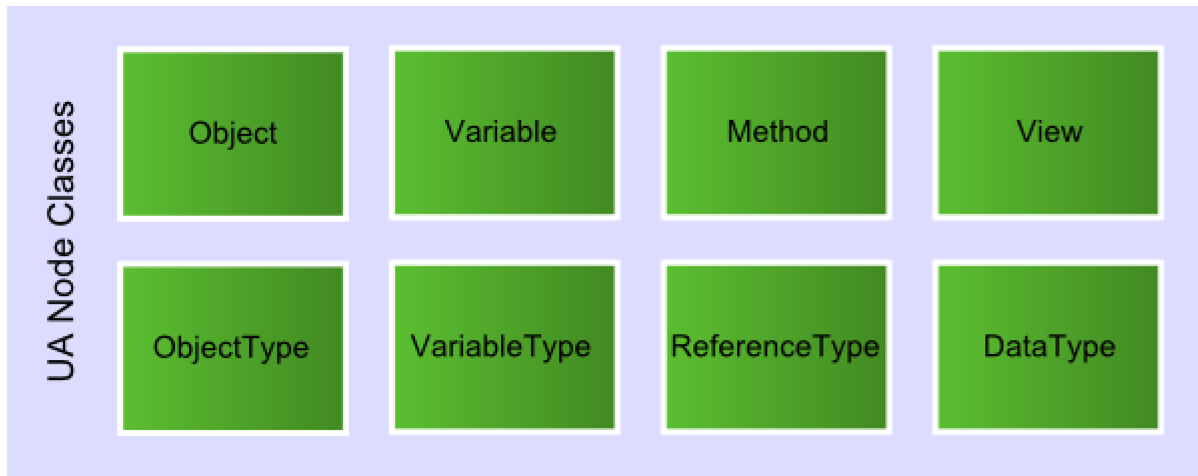


Figure 19: OPC UA Node Classes [4].

The Node Classes defined to represent Objects fall into three categories: those used to define instances, those used to define types for those instances and those used to define data types [9].

3.5.2.1 Base Node Class

All Node Classes are derived from a Base Node Class.[9] The base node class cannot be used directly. It specifies the attributes inherited by all node classes. In all Node Classes, certain Attributes will have a Mandatory or Optional use, meaning some attributes must be present at all instances of a Node Class or can be absent, depending on the application requirement. The most significant attributes are NodeId,

NodeClass, BrowseName, DisplayName and Description. Their main role of these attributes is to assign identity to the nodes and describe what is their purpose. NodeId is used to unambiguously identify a Node in an OPC UA server and is used to address the Node in the OPC UA Services [9]. NodeClass tells the node class a Node belongs to such as Object, Variable or Method.[9] BrowseName identifies the Node when browsing the OPC UA server [9] It is not localized [9]. DisplayName Contains the Name of the Node that should be used to display the name in a user interface [9]. Therefore, it is localized. Description, as the name suggests, contains a localized textual description of the Node.[9]

3.5.2.2 Object Node Class

As mentioned before, The Object node class is used to represent systems, system components, real-world objects and software objects.

Table 2 Object node class specific attributes [4]			
Attributes	**Use**	**Data Type**	**Description**
EventNotifier	Mandatory	EventNotifierType	Indicate if the Node can be used to subscribe to Events or the read / write historic Events

3.5.2.3 Variable Node Class

The Variable node class is used to represent the content of an Object. Variables provide real data and thus can contain a high number of attributes [9]. The useful attributes of a Variable node class, which we need to understand are shown in Table 2.

Table 3 Variable node class specific attributes [4]			
Attributes	**Use**	**Data Type**	**Description**
Value	Mandatory	BaseDataType	The actual value of the Variable
DataType	Mandatory	NodeId	The data type of the value
ValueRank	Mandatory	Int32	Specifies the dimensions of the array, in case the value is an array
ArrayDimensions	Optional	UInt32[]	Specifies the size of the array in each dimension
AccessLevel	Mandatory	AccessLevelType	A bit mask indicating if value is readable and writable and whether the history of the value is readable and changeable
UserAccessLevel	Mandatory	AccessLevelType	Defines additional user access rights

MinimumSamplingInterval	Optional	Duration	Defines how fast the OPC UA server can detect changes of the Value Attribute
Historizing	Mandatory	Boolean	Indicates if value history is recorded

ValueRank may have the following values:

- n > 1: the Value is an array with the specified number of dimensions.
- OneDimension (1): The value is an array with one dimension.
- OneOrMoreDimensions (0): The value is an array with one or more dimensions.
- Scalar (−1): The value is not an array.
- Any (−2): The value can be a scalar or an array with any number of dimensions.
- ScalarOrOneDimension (−3): The value can be a scalar or a one dimensional array.

This information would be useful for us later when designing the information model.

3.5.2.4 Method Node Class

The Method node class is used to represent a Method in the server address space. The following attributes in Table 3 are specific for the Method node class.

Table 4 Method node class specific attributes [4]

Attributes	Use	Data Type	Description
Executable	Mandatory	Boolean	Indicates if the Method can be called
UserExecutable	Mandatory	Boolean	Indicates if User can call the Method

3.5.2.5 ObjectType Node class

The ObjectType node class is used to represent a type node for objects in the server address space [9]. ObjectTypes are similar to a classes in OOP.

Table 5 ObjectType node class specific attributes [4]

Attributes	Use	Data Type	Description
IsAbstract	Mandatory	Boolean	Indicates whether the ObjectType is concrete or abstract and therefore cannot directly be used as type definition

3.5.2.6 VariableType Node Class

Type node for Variables in the server address space are represented by the VariableType node class. VariableType are typically used to define which properties are available on the Variable instance.[9] Table 6 lists the attributes inherited by the instances of the particular Variable Type.[9]

Table 6 Variable node class specific attributes [4]

Attributes	Use	Data Type	Description
Value	Mandatory	BaseDataType	The default Value for instances of this type.
DataType	Mandatory	NodeId	NodeId of the data type definition for instances of this type
ValueRank	Mandatory	Int32	Specifies the dimensions of the array, in case the value is an array
ArrayDimensions	Optional	UInt32[]	Specifies the size of the array in each dimension
IsAbstract	Mandatory	Boolean	Indicates whether the ObjectType is concrete or abstract and therefore cannot directly be used as type definition

3.5.2.7 DataType Node Class

All DataTypes are represented as Nodes of the NodeClass DataType in the Address Space. [9]

3.5.3 NodeId

In OPC UA, every entity in the address space is a node. NodeId is a built-in DataType used to uniquely identify a Node within a Server [9]. The namespace is used to ensure unique NodeIds even if different naming authorities use the same identifiers. This happens if naming authorities work independent of each other like different information model working groups. The NodeId is composed of three elements shown in Table 7.

Table 7 NodeId structure [4]		
Name	**Type**	**Description**
NamespaceIndex	UInt16	The index for a namespace URI
IdentifierType	Enum	The format and data type of the identifier
Identifier	Depends on identifier type	The identifier for a node

Figure 21 illustrates the NodeId contents for different NodeId types.

NodeId	
NamespaceIndex	2
IdentifierType	numeric
Identifier	5001

NodeId	
NamespaceIndex	2
IdentifierType	opaque
Identifier	M/RbKBsRVkePCePcx24oRA==

NodeId	
NamespaceIndex	2
IdentifierType	string
Identifier	MyTemperature

NodeId	
NamespaceIndex	2
IdentifierType	GUID
Identifier	09087e75-8e5e-499b-954f-f2a9603db28a

Figure 21: Examples of different NodeId types [4].

3.5.3.1 NamespaceIndex

The namespace is a URI that identifies the naming authority responsible for assigning the identifier element of the NodeId [9]. Naming authorities include the local Server, standard working group like the OPC UA working group, standards bodies and consortia [9]. Using a namespace URI allows multiple OPC UA Servers attached to the same underlying system to use the same identifier to identify the same Object. This enables Clients that connect to those Servers to recognize Objects that they have in common. The numeric values used to identify namespaces correspond to the index. They are stored in the so-called namespace array also referred to as namespace table [9].

An example of Namespace URI is OPC UA namespace: “http://opcfoundation.org/UA/”. Its namespace index is 0.

3.5.3.2 IdentifierType

IdentifierType defines the format and data type of the identifier. It can be a numeric value, a string, a globally unique identifier (GUID), or an opaque value (a namespace specific format in a ByteString) [9]. Which type is preferred depends on the use case. If it is important to save memory or bandwidth, it makes sense to use numeric NodeIds which are smaller and faster to resolve. The OPC UA namespace, as defined by the OPC Foundation, uses numeric NodeIds [4]. System-wide and globally unique identifiers allow clients to track Nodes, e.g. work orders, moving between OPC UA servers as they progress through the system.

3.5.3.3 Identifier

The identifier is used within the context of the first three elements to identify the Node. We will be using the value of the identifier to link static or dynamic values and method call to specific Nodes.

4. Task motivation and solution

4.1 Motivation

IO-Link devices are used widely in the industrial automation sector and access to these devices is mostly limited to the shop floor. There is a great demand for a standardized and manufacturer independent interface for accessing IO-Link data remotely through cloud services or exchanging over different networks. OPC UA, over several years, has proved to be the ideal candidate for providing remote access to IO-Link devices. Common use cases OPC UA support for IO-Link would include:

- Parametrization of IO-Link Devices connected to IO-Link Master
- Subscribing to master or device variables and events
- Reading product identification
- Reading diagnostics data
- Supervision of plant and machine status

Because the communication of OPC UA is based on TCP/IP it is compatible with many Ethernet-based fieldbuses such as PROFINET. The small memory footprint allows OPC UA Server applications to be installed on PLCs or directly on IO-Link Master devices.

4.2 Solution proposal

In order for OPC UA to interface and access the functionality in IO-Link devices, we would use the IO-Link Master API provided internally in SIEMENS. The API contain functions which encapsulate

UART/SPI communication of the IO-Link modules. The Application Layer and System Management of IO-Link shown in Figure 2, provide functions necessary for handling exchange of data and port configuration, respectively.

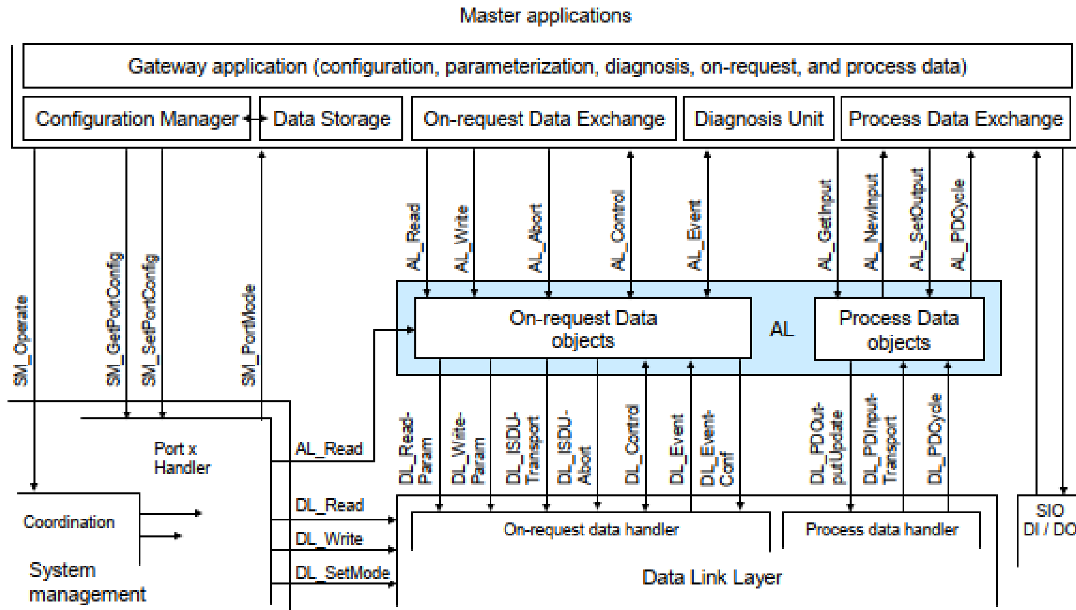


Figure 11: Structure and services of the application layer in IO-Link Master [3].

The goal would be to map the service of OPC UA such as Read, Write, Call with their corresponding IO-Link API counterparts, by creating custom service handlers. Each individual data from IO-Link should be mapped with their corresponding node in accordance with the Companion Specification [2]. At the time of working on this project, the restrictions imposed due to the pandemic situation worldwide limited the access to the IO-Link Master hardware platform using the IO-Link API, as a result we had to resort to using the Raspberry Pi platform, as an IO-Link Master instead. We will use the OPC UA information modelling framework to create an Address Space which will represent the IO-Link devices as Objects and implement the use cases for reading cyclic process data from a variable and invoking an IO-Link service. The process data will be presented as value store and IO-Link service will be handled as a method call by OPC UA, asynchronously.

For the task of implementing the solution and testing it we will use several software components:

1. OPC UA High Performance SDK version 1.4.1.263 licensed - for the development of server application running on the hardware.
2. UaModeler version 1.6.3.454 unlicensed – for designing information model for IO-Link devices and generating relevant server application source code files.

3. SiOME version 2.0.4 – Siemens OPC UA Modeling Editor for designing information model for IO-Link devices and export as xml file.
4. UaExpert version 1.5.1.331 – OPC UA client application for accessing the server application.

4.2.1 Task steps

The task could be generalized in several steps:

- Creating and generating the information model for IO-Link
- Generating the server application
- Implementing the value-store for reading dynamic data from a variable
- Implementing a method for calling from client
- Cross - compiling the application for Raspberry Pi
- Connecting to server running on RPi and verifying functionality

5. OPC UA SDK

An integral part of the of the project is the use of OPC UA SDK which provides the base for our application in terms of secure communications and functionality, as well as adaptation for different platforms. An OPC UA SDK reduces the development effort and facilitates faster interoperability for an OPC UA application. The SDK used in the project is High Performance SDK version 1.4.1.263 for Linux. It is a commercially licensed product distributed by Unified Automation. The documentation provided by UA, which is available online [4], provides detailed information on the individual components of the SDK, their purpose and how they work. It is essential to understand these components, as well as examine the example applications codes, in order to build applications and integrate IO-Link into OPC server.

5.1 IO-Link Information model

In order to read and write process data, send commands and configure IO-Link devices using OPC UA, we need to create a set of Nodes that will represent the IO-Link entities and their components in the Address Space of OPC UA. We will follow the "OPC UA for IO-Link Companion Specification" defined by IO-Link Community Consortium [2], to create the IO-Link Information model. We will use the Siemens OPC UA Modeling Editor tool, provided internally in Siemens, to create the Nodes and additionally use the UaModeler to generate source files for our server application. In this project we used an unlicensed version of UaModeler available by UA through registration in their website. The unlicensed version does not allow us to export the new information model containing more than 10 nodes, in XML format and for that reason we are used SiOME. As the source files generated by UaModeler are not going

to be used in a real commercial product and is simply for research and demonstration purposes, there should not be any legal issues.

5.1.1 Model Overview

Figure 22 gives an overview of the IO-Link Information Model. The IOLinkDeviceType represents IO-Link Devices. This this type shall directly be used to represent an IO-Link Device if no IODD file is available, otherwise IOLinkIODDDeviceType is used [2]. The IOLinkDeviceType inherits from TopologyElementType defined in OPC UA Part 100 [12] and thus provides basic grouping mechanisms, such as ParameterSet for parameters and MethodSet for Methods. It also provides basic Properties of a device like SerialNumber, VendorId, ApplicationSpecificTag and Methods like ReadISDU, defined in OPC UA Part 100. The IO-Link Master is represented by an Object of IOLinkMasterType. This ObjectType also inherits from the TopologyElementType. For each port the IO-Link Master contains an Object of type IOLinkPortType. The IOLinkPortType inherits from the TopologyElementType and thereby uses the same grouping mechanisms for Parameters and Variables. If the port has an IO-Link Device connected, the Object of type IOLinkDeviceType is connected to the port.

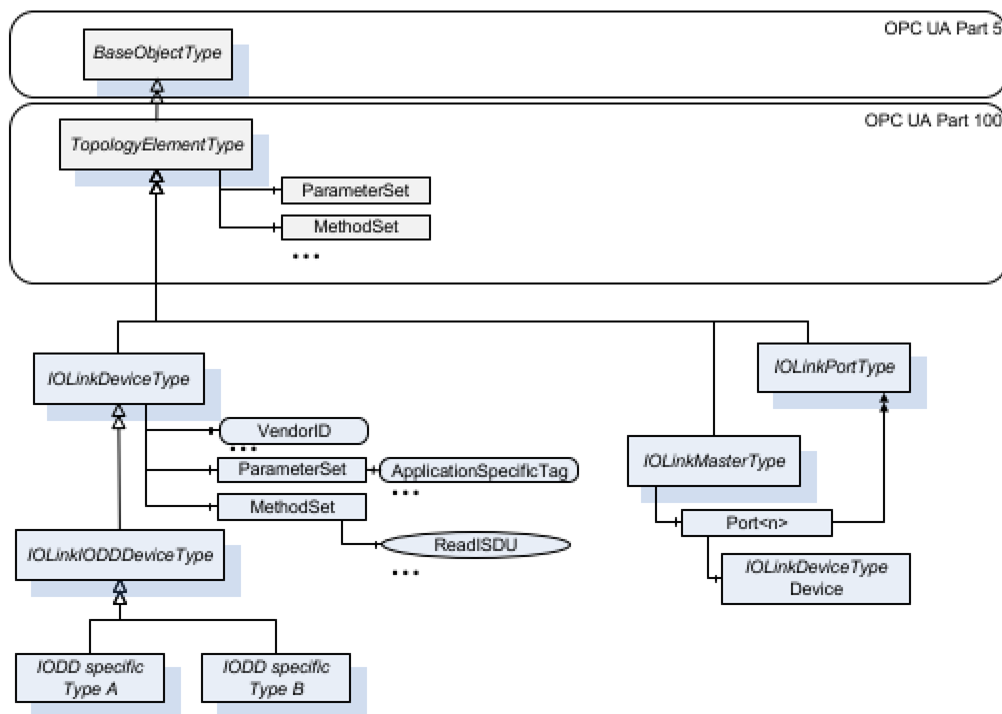


Figure 22: IO-Link Information Model overview [22].

Figure 22 uses a notation that was developed for the OPC UA specification and in order to understand it we can refer to Figure 23 that describes it.

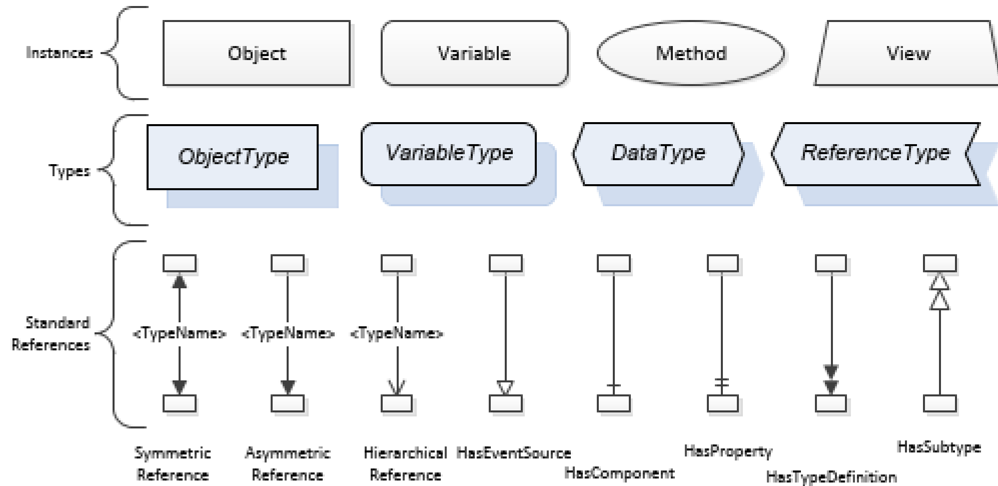


Figure 23: OPC UA Information Model Notation [2].

5.1.2 SiOME

"Siemens OPC UA Modelling Editor" (SiOME) tool, is an editor that allows us to define your own OPC UA information models or mapping existing companion specifications on SIMATIC PLC / SINUMERIK [11]. The Figure 22 illustrates the interface of the SiOME.

SiOME offers the following functions which facilitate the generation of information models:

- Import predesigned OPC UA companion specifications.
- Saving the work status (project) in XML format and re-import.
- Modeling of own types, objects and methods.
- Comprehensive access monitoring by setting the access rights.

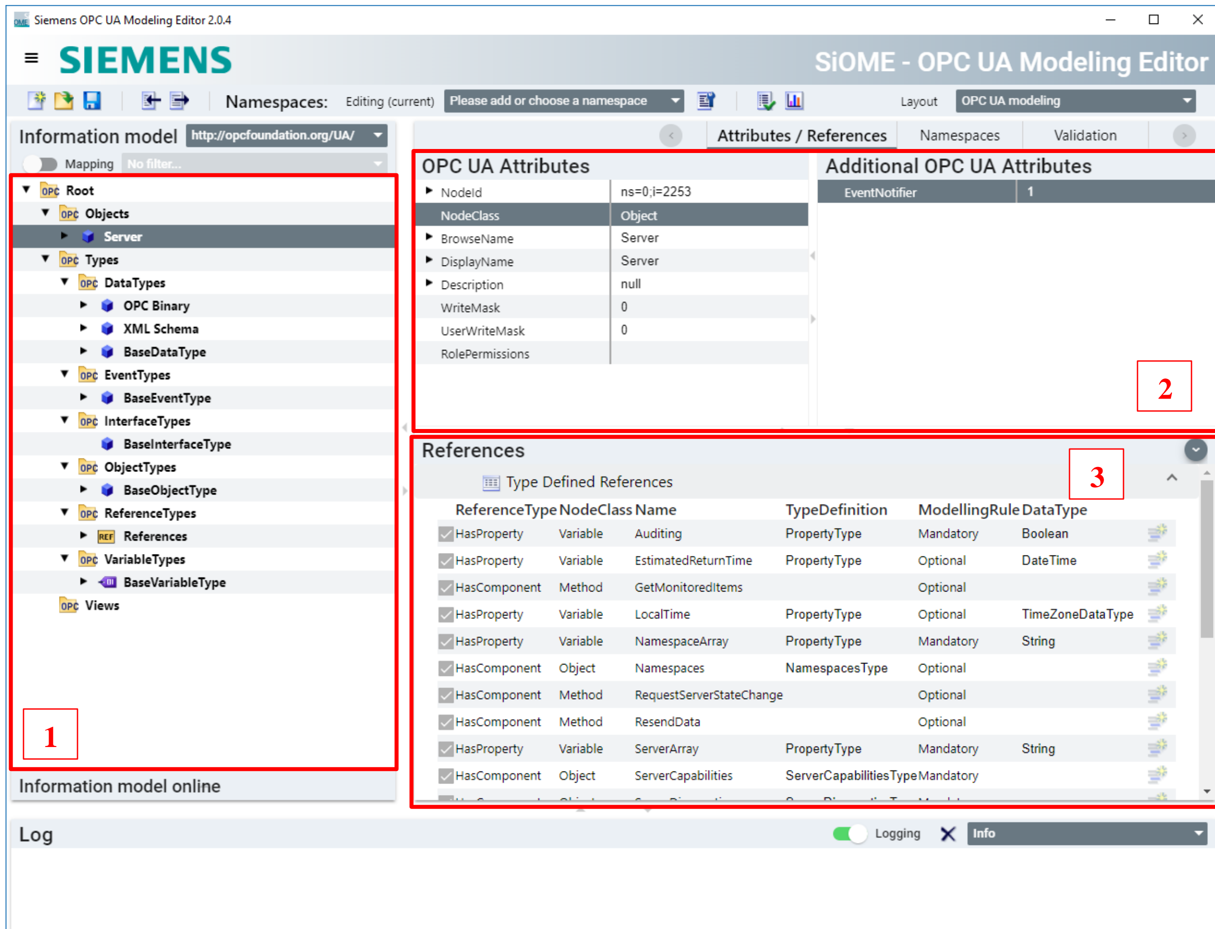


Figure 22: SiOME interface.

After starting up SiOME, we are met with different working elements of the editor. The important areas that are useful for us Information Model, Attribute and References areas outlined in red in Figure 22, numbered 1,2 and 3, respectively. Information Model area shows us the complete model that consists of of all NodeSets that are imported from their corresponding namespace. By default, SiOME comes with Base NodeSet and DI NodeSet preloaded, so we are will build our model using those NodeSets.

The aggregated information model is represented as tree view for navigation. Different Node types are located in Types subfolder and are organized within their own subfolders. The Object folder contains the instances of those Node Types. When creating a new project, an instance of ServerType Node is already included in the Object folder, as it will contain the Server configuration parameters.

5.1.2.1 New Project

We begin creating our IO-Link model by clicking on the “New” icon. By default, SiOME loads the OPC UA Base Model only, but since we also need DI model, we need to import the DI XML NodeSet file.

This is done simply by clicking “Import XML” button, navigating through file explorer to the required file directory and adding it.

5.1.2.2 Creating a new Subtype

In this section I will demonstrate the procedure to create a new Subtype from existing Type. As an example, I will create a new Subtype from TopologyElementType which is the parent ObjectType of our IO-Link entities. Right clicking on TopologyElementType give us drop-down menu, we then click “Add New ObjectType”, we then get the option to select which namespace this new ObjectType will belong to. In our case we click on “Create new Namespace” and define the URI as “http://siemens.de/UA/IOLink”. It is not important what URI we define, ass it is only for demonstration purposes.

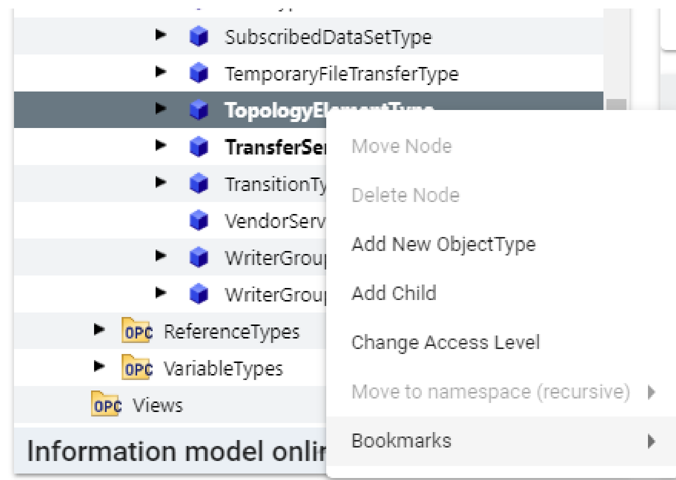


Figure 23: Creating new subtype in SiOME.

5.1.2.3 Adding Optional components

If the parent Object contains components with Optional ModellingRule, we can include those components in our newly created ObjectType from Section 3.8.2.2, by clicking the check boxes in the References area. As an example, in Figure we check MethodSet and ParameterSet, that will group Method and Prameter Nodes.

	ReferenceType	NodeClass	Name	TypeDefinition	ModellingRule	DataType
☐	HasComponent	Object	<GroupIdentifier>	1:FunctionalGroupType	OptionalPlaceholder	
☐	HasComponent	Object	Identification	1:FunctionalGroupType	Optional	
☐	HasComponent	Object	Lock	1:LockingServicesType	Optional	
☑	HasComponent	Object	MethodSet	BaseObjectType	☐ Mandatory	
☑	HasComponent	Object	ParameterSet	BaseObjectType	☐ Mandatory	

Figure 24: Optional components in SiOME..

5.1.2.4 Adding Child elements

In order to add any child elements to a parent node we right click on the parent node and click “Add Child”. This will open a menu, where we can name our child element, define its NodeClass, select the namespace it will belong to, define the ReferenceType and TypeDefinition. As an example, if we need to add a variable to a DeviceType Object, which describes a certain property of that Device Object, such as a Serial Number of device or Manufacturer, in string form, then we make the following selections in corresponding fields:

- NodeClass – Variable
- ReferenceType – HasComponent
- TypeDefinition – PropertyType
- DataType – String

Figure 25: Adding child node in SiOME.

5.1.2.5 Modifying Attributes

Furthermore, in case we need adjust any attributes of an element we can do so in the Attributes area. As an example, we can specify the AccessLevel of our Variable example created in Section 3.8.2.4, change its DisplayName or add a Description.

OPC UA Attributes	
NodeId	ns=2;i=1024
NodeClass	Variable
BrowseName	2:ExampleVariable
DisplayName	ExampleVariable
Description	null
WriteMask	0
UserWriteMask	0
RolePermissions	

Additional OPC UA Attributes	
Value	
DataType	String
ValueRank	1 Dimension
ArrayDimensions	[1]
AccessLevel	3
UserAccessLevel	3
MinimumSamplingInterval	0
Historizing	false

Figure 26: Attributes view SiOME.

5.1.3 Creating IOLinkDeviceType in SiOME

The companion specification [2] provides us the definition for IOLinkDeviceType that represents the generic information of an IO-Link Device and is formally defined in Table 8. We use the operations described in the SiOME section, to construct the elements listed in Table 8. Some elements contain more sub-elements, which are all defined in [2].

Table 8: IOLinkDeviceType definition [2].

Attribute	Value				
BrowseName	IOLinkDeviceType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of TopologyElementType defined in OPC UA Part 100.					
HasComponent	Object	2:ParameterSet		BaseObjectType	Mandatory
HasComponent	Object	2:MethodSet		BaseObjectType	Mandatory
HasComponent	Object	2:Identification		FunctionalGroupType	Mandatory
HasComponent	Object	General		FunctionalGroupType	Mandatory
HasProperty	Variable	2:SerialNumber	String	PropertyType	Optional
HasProperty	Variable	2:Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	2:Model	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	2:HardwareRevision	String	PropertyType	Optional
HasProperty	Variable	2:SoftwareRevision	String	PropertyType	Optional
HasComponent	Variable	2:DeviceHealth	DeviceHealthEnum	BaseDataVariableType	Optional
HasProperty	Variable	MinCycleTime	Duration	PropertyType	Mandatory
HasProperty	Variable	RevisionID	String	PropertyType	Mandatory
HasProperty	Variable	VendorID	UInt16	PropertyType	Mandatory
HasProperty	Variable	DeviceID	UInt32	PropertyType	Mandatory
HasProperty	Variable	DeviceAccessLocks	UInt16	PropertyType	Optional
HasProperty	Variable	ProfileCharacteristic	UInt16[]	PropertyType	Optional
HasProperty	Variable	VendorText	String	PropertyType	Optional
HasProperty	Variable	ProductID	String	PropertyType	Optional
HasProperty	Variable	ProductText	String	PropertyType	Optional
HasComponent	Object	Alarms		FolderType	Optional
GeneratesEvent	ObjectType	IOLinkDeviceEventType	Defined in 9.3.		
GeneratesEvent	ObjectType	IOLinkDeviceAlarmType	Defined in 9.8		

The specification also gives information on how IO-Link Device values are mapped to individual OPC UA elements of IOLinkDeviceType. For example, according to the specification the Variable SerialNumber of DataType String should be mapped to ISDU Index 0x0015 (Serial Number) and if the device does not support this ISDU Index, the Variable shall not be provided [2]. In the scope of this project we are not concerned about mapping, however it will be applied in future development.

Furthermore, we use the definitions for IOLinkMasterType and IOLinkPortType that represent the generic information of an IO-Link Master and IO-Link Port, respectively, to create them in SiOME. In physical domain, an IO-Link Master has a port to which an IO-Link Devices may or may not be connected. Therefore, IOLinkMasterType will contain a Mandatory Object Node of IOLinkPortType and IOLinkPortType will contain an Optional Object of IOLinkDeviceType. Depending on whether an IO-Link Device is connected to the port the instance IOLinkDevice will be present or absent in an instance of IOLinkPort [2].

The resulting Information Model will now contain our new IO-Link Type Nodes as shown in Figure 27.

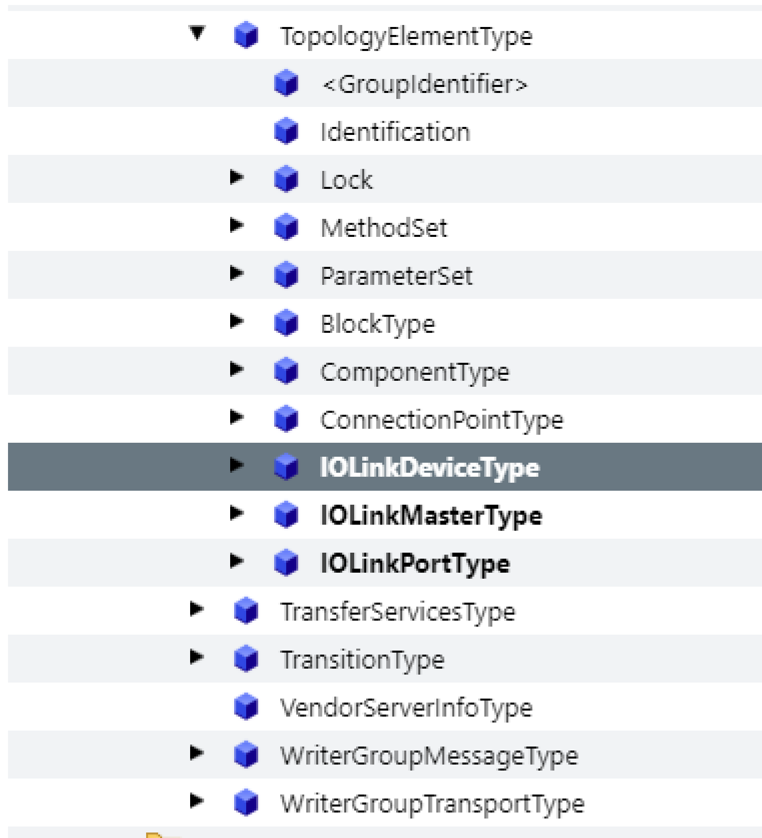


Figure 27: IO-Link type nodes created in SiOME.

Before, we export our Information Model as XML file, we can create an instance of IOLinkMasterType with IOLinkPort Object within it enabled, as well as IOLinkDevice within IOLinkPort also enabled. The instaces of these Nodes will be static within the Information Model, meaning already present after initializing the model in the server application.

5.1.4 Generating code files using UaModeler

The UaModeler is a specialized tool which complements the SDK, that is also distributed by UA. Not only it simplifies the modeling of information itself, but additionally it generates the source code required to implement the designed model and provides graphical design of the address space. The generated code exactly fits into the related SDK. The UaModeler can generate code for C++, ANSI C, .NET and High-Performance SDKs. For generating code, the UaModeler uses a template-based code generator. Several templates are combined to template sets which are responsible for the generated code [13].

The GUI of UaModeler is shown in Figure 28.

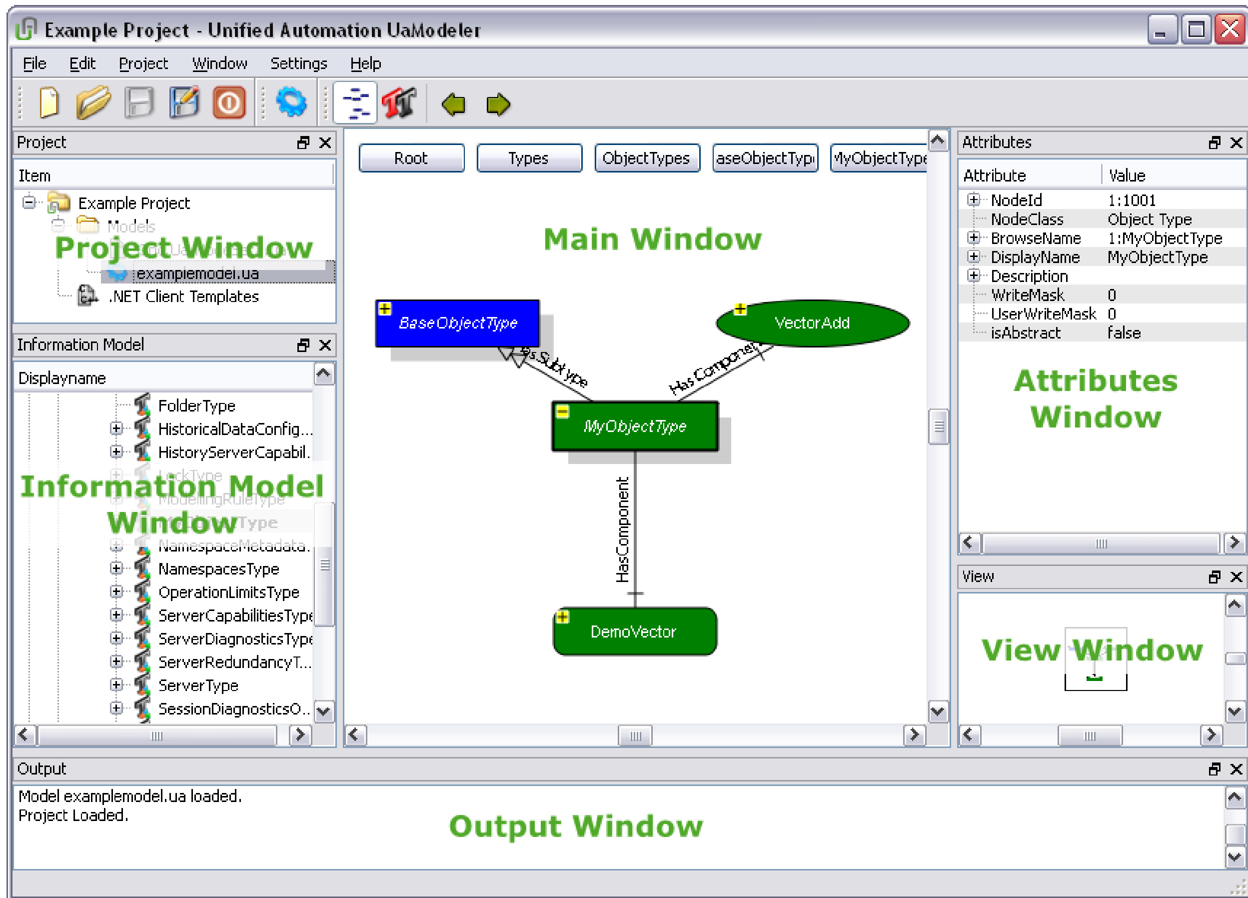


Figure 28: UaModeler GUI [13].

The GUI has a similar environment compared to SiOME. In order to generate the source files we need, we begin by creating a new project. It is important to select the correct template sdk and version, to avoid compatibility issues. Current version of UaModeler provides a template for HP SDK version 1.3 or higher [13]. After selecting the template, we proceed to choosing the base models we will use, in our case we select the DI model (UA Base model is selected by default). We can finalize the set up without any additional input. After the project is created, we will remove the new model from Models folder in the Project Window and replace it with our model generated by SiOME. We right click on the Models folder, click “Add Existing Model” and import the XML file. We will modify the model settings of the DI model and our IOLink model. We will set the Library and Prefix of DI model as “di” and our IOLink model as “iolink”. That way our generated files will be labeled properly. Before, generating files we will enable code generation for the DI model, by right clicking on the model and “Generate Namespace”.

The Template Set for the High-Performance SDK creates the following files:

- Identifiers for static nodes
- Structures for structured DataTypes, including encode and decode functions:

    - Normal structured DataTypes (without optional fields)
    - Structured DataTypes with optional fields
    - Unions
- Enumerations for enumerated DataTypes
- Binary files for the addressspace
- Functions for method handling:
    - Checking input arguments
    - Method stubs for simple method implementation
- A default provider
- A default server application
- A CMakeFile for the application

5.1.5 Generating dynamic address space binaries

Even though UaModeler is able to automatically generate the address space binaries, we need to do it manually. The SDK contains Linux and Windows versions of xml2bin and xml2c tools. The Linux version of the tools do not function properly whereas the Windows version does. The UaModeler for Windows invokes the linux version of the tools and could possibly be due to a present software bug, that is yet to be patched.

The commands that are executed to generate the address space binaries are as follows:

```
> xml2bin -i0 -o ns0.bin Opc.Ua.NodeSet2.xml

> xml2bin -i2 -o Opc.Ua.Di.NodeSet2.bin Opc.Ua.NodeSet2.xml Opc.Ua.Di.NodeSet2.xml

> xml2bin -i3 -o iolink_model.bin Opc.Ua.NodeSet2.xml Opc.Ua.Di.NodeSet2.xml iolink_model.xml
```

The commands are executed in Windows command line. It does not matter which version of the tools we use as the information models are in binary format and can be loaded in any OS environment.

5.2 Building application in SDK environment

The building of the OPC UA stack and our application needs to be performed in a Linux environment, therefore we run Debian 10 Linux OS in a virtualized form. For virtualization we use Virtual Box.

There are two approaches to building our application. It is possible to build our application outside the SDK directory or inside the directory. I found it was easier and less error prone to build with our application files located inside the SDK, thus avoiding the need to manually configure the link to the

libraries and set up necessary environment variables. The application folder was copied over to the "examples" directory in the SDK and was added to the build list in the CMake file in the "examples". This would allow our application and the example applications provided by the SDK to be compiled altogether.

5.2.1 Integrating into CMAKE file

In order for the cmake to include our project folder, we need to edit the CMakeLists.txt file located in the "examples" folder. First we use the *cmake_dependent_option()* to create an internal option for our project that allows us to enable or disable the build of our project or disables it automatically if the building procedure is disabled for the whole "examples" folder or the server SDK library. The bottom line shown in Figure 29 was added to the CMakeLists.txt file in "examples" folder.

```
cmake_dependent_option(BUILD_SERVER_DEMO          "Set to ON to build uaserver_hp" ON "BUILD_EXAMPLES;BUILD_LIBUASERVER" OFF)
cmake_dependent_option(BUILD_SERVER_MINIMAL       "Set to ON to build uaserver__minimal" ON "BUILD_EXAMPLES;BUILD_LIBUASERVER" OFF)
cmake_dependent_option(BUILD_SERVER_DI            "Set to ON to build uaserver_di" ON "BUILD_EXAMPLES;BUILD_LIBUASERVER" OFF)
#added line
cmake_dependent_option(BUILD_SERVER_IOLINKMODEL   "Set to ON to build uaserver_iolink" ON "BUILD_EXAMPLES;BUILD_LIBUASERVER" OFF)
```

Figure 29: CMake dependent option for our project.

Next, we insert a condition which will add our project directory into the build process if its corresponding CMake option is enabled. Figure 30 shows the added condition, opening with an *if()* keyword, performing the command *add_subdirectory(),* and closing with the keyword *endif()* .

```
# added condition
if (BUILD_SERVER_IOLINKMODEL)
    add_subdirectory(iolink_model)
endif()
```

Figure 30: Condition for adding project folder into build.

There is no specific reason for choosing the "examples" directory as the parent directory for our project and it simply serves as an example of integrating our project into an existing directory's CMakeLists.txt file.

5.2.2 Adding Value Store

In order to demonstrate the reading, writing and updating of values of arbitrary Variable Nodes, we will implement a Value Store into our application. These are integrated in the SDK and offer easy access to values. The implementation of a value store to access a device is much simpler than implementing service handlers for read, write, and subscription, but the value store is limited to devices with synchronous access to the data [4]. Our IOLink provider will use the Value Store to store and access values of corresponding Nodes in the IOLink Address Space.

The Value Stores have an associated store index, which is used to identify instances of Value Store and a value index, used to identify a value inside a store. Both indices are saved in variable nodes, and the SDK uses them find the associated store for the node and retrieve or write a value.

To implement our Value Store, we must declare an interface for our store and register it at the global Value Store management [4]. The registration and variable node linkage to our Value Store is done during provider initialization in *provider_iolink.c,* while the value array and accessor functions passed to the registration are defined in *provider_iolink_store.c.*

First, we define an index for our custom store as a global variable. After loading and registering the IOLink namespace, we create our Value Store the following way:

```
struct ua_valuestore_interface store_if;

/* register custom store */
ua_valuestore_interface_init(&store_if);
store_if.get_fct    = iolink_store_get_value;
store_if.attach_fct = iolink_store_attach_value;
ret = ua_valuestore_register_store(&store_if, &g_custom_store_idx);
if (ret != 0) return ret;
```

We declare a store interface as a *ua_valuestore_interface* type structure and assign our getter and setter functions and register the interface using *ua_valuestore_register_store,* which accepts the requested store index as a second parameter. In our case, it is zero, meaning the SDK will assign a free index [4].

We add the line *ua_valuestore_register_store* to the cleanup process to ensure the memory for our custom store is cleared properly to avoid memory leaks.

In order to demonstrate and verify the functionality our Value Store implementation, we will write the store index and a value index to an arbitrary variable node with numeric ID of 1076:

```
node = ua_node_find_numeric(g_provider_iolink_nsidx, (uint32_t)1076);

ret = custom_store_set_initial_value(0, (uint32_t)7777);
if (ret != 0) return UA_NODE_INVALID;

/* write the storeindex to the node */
ret = ua_variable_set_store_index(node, g_custom_store_idx);
if (ret != 0) return UA_NODE_INVALID;

/* write the valueindex to the node */
ret = ua_variable_set_value_index(node, 0);
if (ret != 0) return UA_NODE_INVALID;
```

ua_node_find_numeric will return the node handle of the node defined by its numeric ID. We will initialize the value at index 0 of our Value Store array to value 7777 and pass the array index to our node as the value index. In *provider_iolink_store.c* we define the Value Store array and getter and setter functions that read from and write to the array. The getter function checks if value index is not read out of bounds of the array and reading with index range is not possible as this is a scalar. Then the value and source timestamp are set (if required), and the status code of the result is set to good, unless an error occurs in which case bad status is returned. The parameters of the setter function are similar to the getter function, but instead of the result, the value to write is passed. Furthermore, in case of an error a bad status code must be passed back as return value. The function itself again needs to check the value index and the index range. Then it has to make sure that only the value is written and no timestamp or status code. Finally, the type of the value must be checked, and the value can be written to the corresponding slot of the array. In order to simulate reading cyclic process data, such as measurement values from a sensor, the value of the variable node is set to the 32 byte timestamp every time the getter function (*iolink_store_read*) is invoked. We can later subscribe to the variable node through an OPC Client program and observe the periodic change of the value.

5.2.3 Adding Method

Certain functionality of connected IO-Link devices can be represented as method nodes. By invoking a call service on a method node, it is possible to execute a function registered for that particular node. As a demonstration we will implement a function which will manipulate GPIO pins on the Raspberry Pi Zero. To control the GPIO pins we will use a C library available as a Debian package *libgpiod-dev*. It is installed in our Debian environment through a terminal by using the following command:

> sudo apt-get install libgpiod-dev

This will install static libraries and API headers that we will include in our project. According to the description of the library [14], the library uses file I/O operations which are blocking functions and are prohibited to be used synchronously by the IPC as it may block the main loop of the application. For this reason the synchronous blocking implementation should be implemented asynchronously by spawning a thread [4]. We will initialize the method and register custom service handler for calling methods for our IO-Link provider in *provider_iolink.c*, the *provider_iolink_methods.c* will contain their definitions and *provider_iolink_system_command.c* will contain the method handler and the method implementation.

During the initialization of the IO-Link provider, the method initialization is performed. The IO-Link provider initialization is finalized by registering custom service handler for calling IO-Link methods:

```
/* register method call handler */
ctx->call    = provider_iolink_opc_method_call;
```

The method handlers are represented in *provider_iolink_methods.c* as function pointers and are stored:

```
/* method handler type definition */
typedef ua_statuscode (*provider_iolink_opc_method_t)
                      (struct uaprovider_call_ctx *ctx,
                       const struct ua_callmethodrequest *req,
                       struct ua_callmethodresult *res);

/* method table for iolink namespace */
#define PROVIDER_IOLINK_OPC_MAX_METHODS 5
static provider_iolink_opc_method_t
       provider_iolink_opc_methods[PROVIDER_IOLINK_OPC_MAX_METHODS];
```

Each method handler will accept parameters containing context information, as well as request data passed over to the method, including input values, and response data that will contain the output data attached after method execution. All the method handlers will be stored in an array, that will have a predetermined size.

The *provider_iolink_opc_method_init* function will iterate over an array of type *provider_iolink_method_node* which is a structure containing a method node index and its corresponding method handler:

```
/* method node structure */
struct provider_iolink_method_node {
    unsigned int method_idx;
    enum ua_identifiertype type;
    uint32_t numeric; /* numeric nodeid or index to value */
    provider_iolink_opc_method_t handler;
};

/* array of method node structures */
static const struct provider_iolink_method_node provider_iolink_method_nodes[] =
{
    {0, UA_IDENTIFIER_NUMERIC, 1101, provider_iolink_call_system_command},
};
```

In the code snippet above we used the numeric ID of value 1101, which identifies a method node, and method handler *provider_iolink_call_system_command* that will be registered for that node. The methods are registered to nodes using the SDK function *ua_method_set_index*, by passing the node handler and a method index as parameters:

```
/* register method to node*/
ret = ua_method_set_index(node, method_index);
if (ret != 0) return ret;

/* store the method handler */
provider_iolink_opc_methods[method_index] = methodhandler;
```

When a call service is invoked on a method node, the *provider_iolink_opc_method_call* looks up the method index assigned to the node by using *ua_method_get_index*:

```
ua_node_t method;
unsigned int method_index;

/* get node handler for requested method */
method = ua_node_find(&req->methods_to_call[i].method_id);

/* get method index for corresponding method node */
ua_method_get_index(method, &method_index);
```

After calling the relevant method handler, the *provider_iolink_opc_method_call* is finalized by calling *uaserver_call_complete*. All method handlers consist of 3 sequential parts: checking input arguments, calling the method implementation and attaching the output arguments to the response. In order to check

the input and output arguments we create an array of type descriptions of each input argument which contains the a pointer to the namespace index in case of a complex type the type ID, the variant type, and flags to influence the behavior of the function. The arguments are checked using helper function *provider_iolink_check_arguments:*

```
/* check argument types */
status_code = provider_iolink_check_arguments
              (g_system_command_in_args,
               countof(g_system_command_in_args),
               ctx, req, res);

if (status_code != 0) return status_code;
```

After checking the input arguments, the method implementation is called:

```
status_code = provider_iolink_system_command(
        ctx,
        res,
        &req->object_id,
        &req->input_arguments[0].value.ui8,
        &out1,
        &out2);
```

The operation of the method handler is finalized by attaching the output from the method implementation to the provider call result:

```
/* attach output to result */
status_code = provider_iolink_attach_arguments(
                res,
                g_system_command_out_args,
                countof(g_system_command_out_args),
                &out1,
                &out2);
```

The implementation of the *provider_iolink_system_command* is as follows:

```
ua_statuscode provider_iolink_system_command(
    /*  in */ struct uaprovider_call_ctx *ctx,
    /*  in */ struct ua_callmethodresult *res,
    /*  in */ const struct ua_nodeid *object_id,
    /*  in */ uint8_t *cmd,
    /* out */ uint16_t *errortype,
    /* out */ int32_t *status)
{
    uint32_t ret;

    /* parameters to be ignored by compiler */
    UA_UNUSED(ctx);
    UA_UNUSED(res);
    UA_UNUSED(object_id);

    *errortype = 0;
    *status = -1;

    ret = system_command_async(*cmd, cb);
    *status = ret;

    return ret;
}
```

Some of the input parameters are not used in this implementation, such as *ctx*, *res* and *object_id,* which are context for the provider call, method result container and the node ID of the parent node, respectively. The input parameter *cmd* is a pointer that will contain the pin number, which will be passed to the asynchronous implementation of the GPIO function, as well as a callback function, for debugging purposes. The *system_command_async* is defined in the following code:

```c
int system_command_async(uint8_t pin, void (*callback)(int error))
{
    struct command_context *ctx;
    pthread_attr_t attr;
    int ret;
    /* create thread context */
    ctx = malloc(sizeof(*ctx));
    if (ctx == NULL) goto memerror;
    ctx->pin = pin;
    ctx->callback = callback;

/* create detached thread */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHABLE);
    ret = pthread_create(&ctx->thread, &attr, gpio_command, ctx);
    pthread_attr_destroy(&attr);
    if (ret != 0) goto threaderror;
    return 0; /* success */
threaderror:
    free(ctx);
memerror:
    return -1;
}
```

We will create information context for *gpio_command*, which will contain information about the thread, pin number and the callback function that will be executed at the end of the thread execution. The code above uses platform specific functions to create the thread. In this case we are using Linux specific functions, which are provided by the SDK. The thread created as detached, which means it will clear out the memory automatically after it terminates. The *gpio_command* run the *gpiod_ctxless_set_value*, which is a high-level function that sets the GPIO pin to either high or low and executes a callback function that after it terminates as defined in the given code:

```c
/* The gpio command thread which uses a blocking function. */
static void *gpio_command(void *arg)
{
    struct gpio_context *ctx = arg;
    int error;

    error = gpiod_ctxless_set_value("gpiochip0",
                                    ctx->pin,
                                    1,
                                    false,
                                    "consumer",
                                    (void (*)(void *))usleep, (void *)5000000);
    /* send callback */
    ctx->callback(error);
    /* cleanup memory */
    free(ctx);
}
```

The GPIO library function accepts the GPIO chip label as its first argument, followed by, pin number, GPIO value to be set, boolean value indicating if active state is when low, name of the consumer and an optional callback [14]. In our case we use a sleep function as our call back, which sets the GPIO pin to high for 5 seconds.

The overall result of this implementation is that the program returns to the caller immediately after the method is invoked, thus not blocking the main loop of the application.

5.3 Cross-compiling for Raspberry Pi Zero

The OPC UA provides a build script for their SDK, which enables the user to compile or cross-compile the SDK using CMake. The main difference is that we need to define the path to the toolchain file by adding the *-DCMAKE_TOOLCHAIN_FILE* parameter to CMake arguments [15]. A toolchain file is a CMake file that configures CMake to use a certain toolchain. The reason for using such file is because all platform specific settings, like system libraries and compiler and linker flags are defined in the toolchain file, so we don't need to include platform specific parts in our projects CMake files (CMakeLists.txt) [15]. We also need to create a target configuration file to configure the build process.

The cross-compiler toolchain with GCC version 8.3.0 was downloaded from [16]. This version was chosen, among several others, with respect to the host OS and target OS. Additionally, precompiled static libraries for *OpenSSL*, which is required for enabling security features, and *libgpiod*, which provides

GPIO control, were imported into the toolchain library folder. These are required for the proper functioning of the application.

5.3.1 Creating target configuration file

The target configuration files are required to be named as *target_configuration_<target_device>*, where *<target_device>* is user defined. In our case it is named *target_configuration_rpizero*. In the file we define several BASH variables, such as the build directory, destination directory, build type and path to toolchain file. The target configuration file will be loaded to the build script during execution.

5.3.2 Creating toolchain CMake file

The toolchain CMake file will define several important CMake parameters such as *CMAKE_SYSTEM_NAME* , *CMAKE_C_COMPILER* and *CMAKE_EXE_LINKER_FLAGS*. *CMAKE_SYSTEM_NAME* is defined as “Linux”, which means Linux-specific platform layer modules will be compiled. *CMAKE_C_COMPILER* will be the path to the C Compiler in the toolchain, and *CMAKE_EXE_LINKER_FLAGS* will tell linker where to look for shared libraries when linking.

5.3.3 Running build script and uploading binaries to the SDK

The build script provides the convenience of running *cmake, make* and *make install*, sequentially, altogether. The command is executed in the terminal as follows:

```
> ./build.sh -p embedded -t rpizero
```

The *-p embedded* option will load a profile, which defines set of CMake parameters appropriate for embedded devices. These parameters can enable or disable certain OPC UA Services, which may not be necessary, and reduce code size as a result. These parameters can additionally be modified by running *cmake-gui* command in the build folder. -t rpizero option will load the target configuration file for the corresponding user-defined target name.

After successfully compiling the project files, the resulting project binaries are uploaded to the Raspberry Pi using *scp* command, which is a command-line tool used to securely copy files between destinations.

5.3.4 Connecting to server using UaExpert and verifying functionality

The Raspberry Pi Zero does not provide a GUI and can only be accessed through a command line, using *ssh* command. After connecting to the device, the project is executed the following way:

```
> ./iolink_model  -d 32
```

The debug level option *-d* is set to 32, which displays the trace information. Running the application will show the output in Figure 31.

```
pi@raspberrypi:~/embedded $ ./iolink_device -d32
N|10|21:20:52.250512|391| uaapplication_load_certificates: matched cert based on config values (store 0)
N|10|21:20:52.253791|391| uaapplication_load_certificates: check if configuration is correct!
N|10|21:20:52.256845|391| uaapplication_load_certificates: change configuration to:
N|10|21:20:52.257714|391| uaapplication_load_certificates:     store://5AE4091F07B71919FB66F91B314126804239890C
N|10|21:20:52.258704|391| uaapplication_load_certificates: loaded cert: store:// (store 0)
N|10|21:20:52.260203|391| uaapplication_load_certificates: certificate structure validated
N|11|21:20:52.343341|391| Registering dynamic address space: http://opcfoundation.org/UA/
N|11|21:20:53.041819|391| Registering dynamic address space: urn:[CompanyName]:iolink_device:raspberrypi
N|11|21:20:53.100922|391| Registering dynamic address space: http://opcfoundation.org/UA/DI/
N|11|21:20:53.149729|391| Registering dynamic address space: http://opcfoundation.org/UA/IOLink/
Server is up and running.
Listening on opc.tcp://raspberrypi:4840
```

Figure 31: Running the OPC UA Server application.

We can connect to the server using UaExpert, an OPC Client provided by UA. The server has to be added by clicking on the “Add Server” button, which opens the menu window shown in Figure 32.

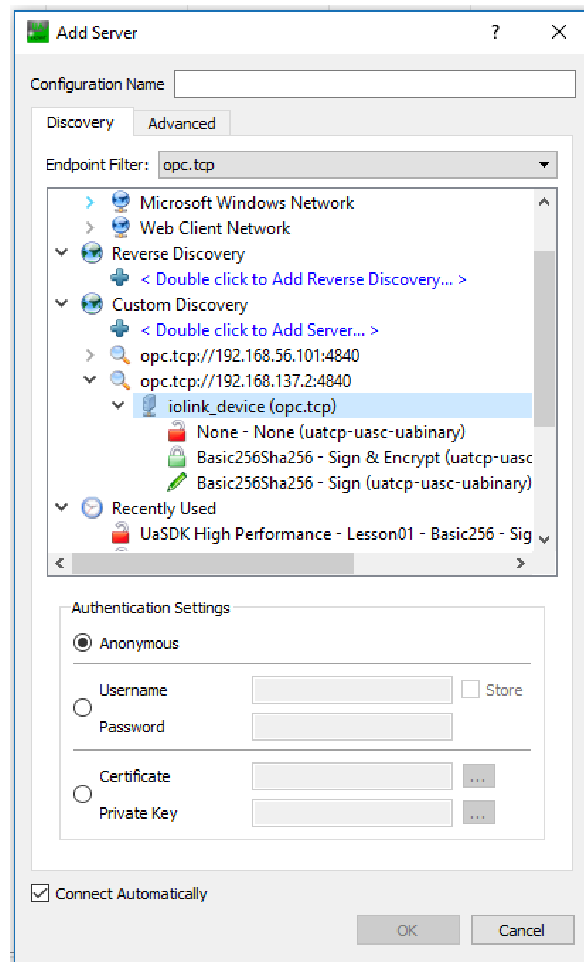


Figure 32: Adding server in UaExpert.

The connection can be established with three different security policies: None, Sign and Encrypt and Sign. When the client is connecting to the server for the first time, it is required to manually accept the Application Instance Certificate in order store the server's certificate for future sessions. The connection window also allows to use username and password to authenticate with the server if they are stored in the server. In this case we connect without user authentication. After connecting to the server, it is possible to traverse the address space in the Address Space window of the client (Figure 33).

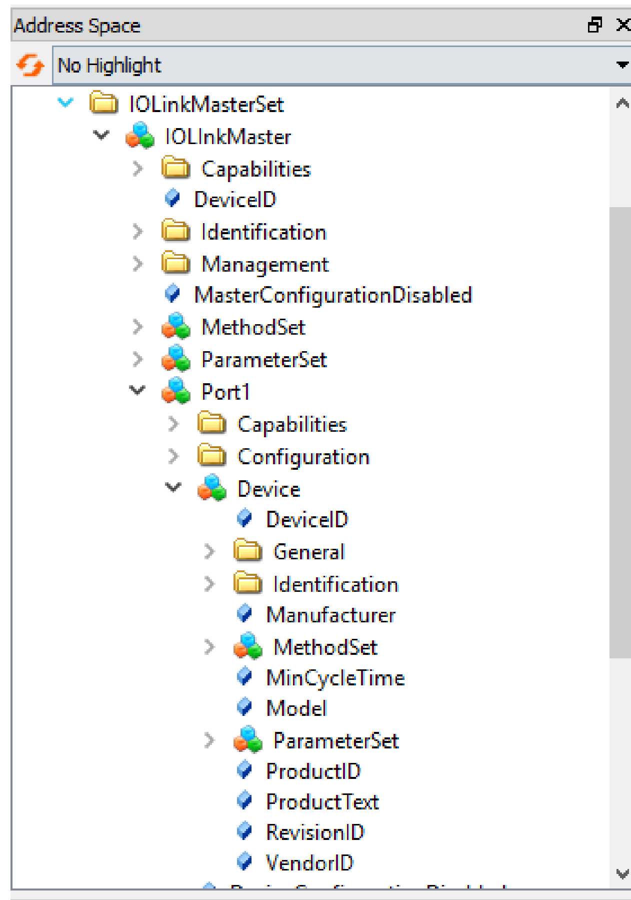


Figure 33: Address Space window.

In Figure 16 we can see the entry point, which is the IOLinkMasterSet folder. It will contain all the IOLinkMaster devices that are connected. Furthermore, Port1 is IOLinkPortType node, which represents the first port of the IOLinkMaster device. Device represents the IOLinkDevice connected to the port. The DeviceID variable node of the Device is the variable node we are using to demonstrate the value store that we had implemented. Clicking on the DeviceID, will display its attributes in the Attributes Window (Figure 34).

Attribute	Value
NodeId	ns=3;i=1076
NamespaceIndex	3
IdentifierType	Numeric
Identifier	1076
NodeClass	Variable
BrowseName	3, "DeviceID"
DisplayName	"", "DeviceID"
Description	"", ""
WriteMask	BadAttributeIdInvalid (0x80350000)
UserWriteMask	BadAttributeIdInvalid (0x80350000)
RolePermissions	BadAttributeIdInvalid (0x80350000)
UserRolePermissions	BadAttributeIdInvalid (0x80350000)
AccessRestrictions	BadAttributeIdInvalid (0x80350000)
Value	
SourceTimestamp	8/11/2020 10:51:39.552 PM
SourcePicoseconds	0
ServerTimestamp	8/11/2020 10:51:39.552 PM
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	905284550
DataType	UInt32
NamespaceIndex	0
IdentifierType	Numeric

Figure 34: Attribute window.

We can see that its numeric ID value is 1076 and it contains some non-zero value. We can subscribe to the node, by drag and dropping it to the Data Access View. The subscription service will poll the value with certain periodicity and display its updated 32 byte value (Figure 35).

Data Access View

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	iolink_device	NS3|Numeric|1...	DeviceID	157440124	UInt32	10:57:34.264 PM	10:57:34.264 PM	Good

Figure 35: Data Access View of subscribed nodes.

In order to verify the functionality of our method, we simply right-click on the “System Command” method and click “Call...” (Figure 36).

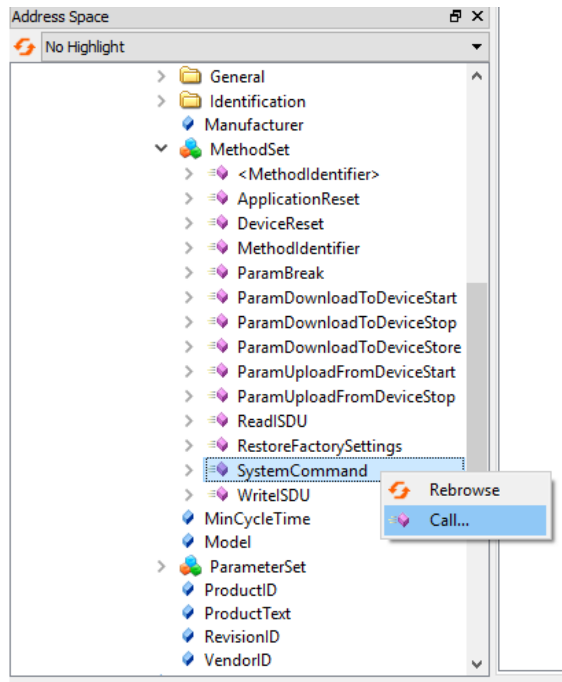


Figure 36: Calling method.

This will open a call window (Figure 37), where we can enter the GPIO pin number as input argument.

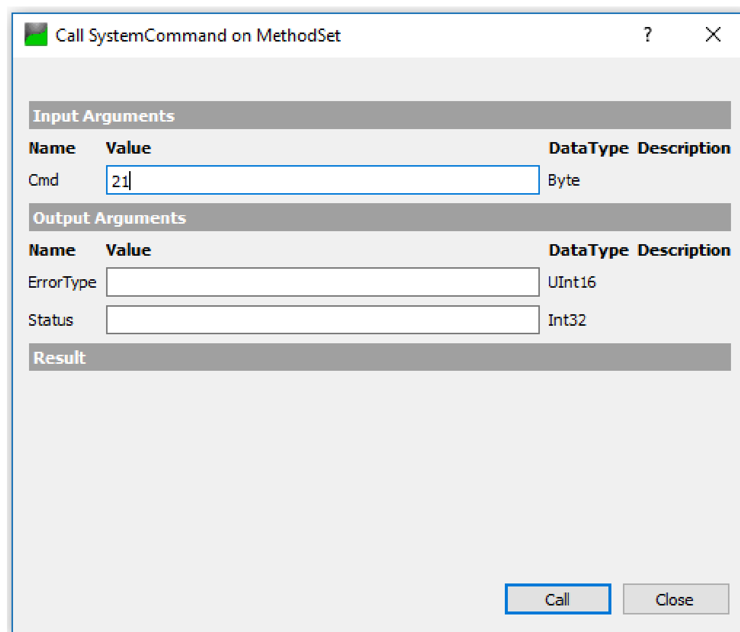


Figure 37: Calling OPC UA method.

This will call the method with the input value and return the output values as shown in Figure 38. In this case the OPC UA will our service call instead of using generic Internal Provider services. The service call

will call the method handler registered for this particular node, which creates a thread that sets the GPIO pin to high for 5 seconds.

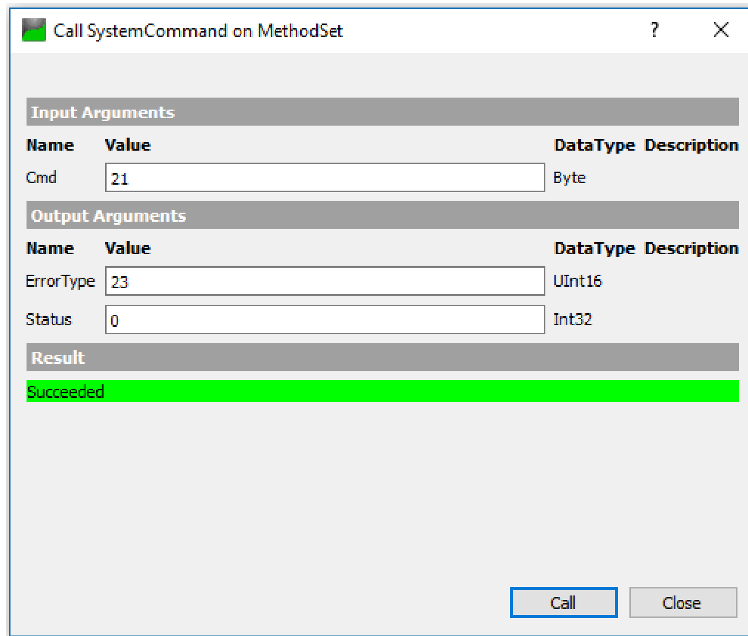


Figure 38: Call result after execution.

The setting of the GPIO pins can be visually verified by the small LEDs connected to the pins (Figure 38). The green LED is connected to pin number 21, while the yellow LED is connected to pin number 27.

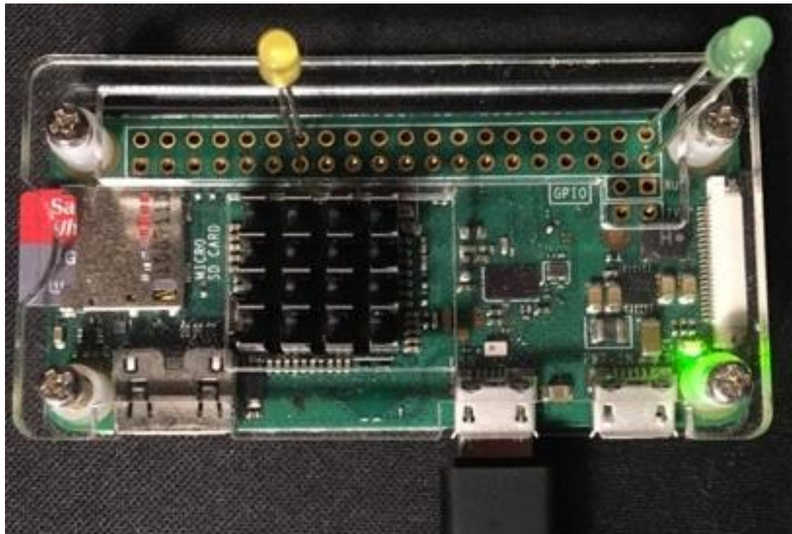


Figure 39: Raspberry Pi Zero with LEDs connected to GPIO pins.

The termination of the thread is also observed by the call back function, which outputs the GPIO exit status in Figure 40 that indicates successful termination of the thread.

```
Server is up and running.
Listening on opc.tcp://raspberrypi:4840
Object ID: 1080
GPIO exit status: 0
```

Figure 40: Output from callback function after thread termination.

The demonstrations above serve only as a basis for handling the access of variables and the handling of calling services that will be applied for connecting IO-Link values to nodes and calling IO-Link functions using the IO-Link API in a similar manner, but on a greater scale.

6. Conclusion

Within the scope of this thesis, we were able to use the modelling framework of OPC UA to create the representation of the IO-Link devices as Objects, abiding the specification provided by IO-Link Community [2], and use the software tools provided by UA to generate code to prototype a working application. The generated application code was thoroughly analyzed, and new code was added create the required functionality for our use case demonstration. The analysis of the OPC UA SDK and necessary tools used to build OPC UA applications gave us insight on how to integrate IO-Link with OPC UA, and demonstrated the functionality for accessing data and calling methods that interact with the GPIO of the Raspberry Pi that acts as an IO-Link Master. Despite the challenges risen throughout the project, the work done serves great contribution for further complete integration of IO-Link with OPC UA and prototyping custom OPC UA applications.

7. References

[1] "IO-Link System Description: Technology and Application" -https://io-link.com/

[2] "IO-Link Community and OPC Foundation: OPC Unified Architecture for IO-Link Companion Specification Release 1.0 December 01,2018" - https://io-link.com/

[3] "IO-Link Interface and System Specification v.1.1.3 June 2019 Order no: 10.002" –

https://io-link.com/

[4] http://documentation.unified-automation.com/uasdkhp/1.4.1/html/index.html

[5] http://documentation.unified-automation.com/uamodeler/1.6.3/html/index.html

[6] http://documentation.unified-automation.com/uaexpert/1.5.1/html/index.html

[7] "OPC 10000-1 - UA Specification Part 1 - Overview and Concepts 1.04" – https://opcfoundation.org/

[8] http://www.ascolab.com/en/technology-unified-architecture.html

[9] "OPC 10000-3 - UA Specification Part 3 - Address Space Model 1.04" - https://opcfoundation.org/

[10] "OPC 10000-4 - UA Specification Part 4 - Services 1.04" - https://opcfoundation.org/

[11]

https://support.industry.siemens.com/cs/attachments/109755133/109755133_SiOME_DOC_V19_en.pdf

[12] "OPC 10000-100 - UA Specification Part 100 - Devices 1.02.02" - https://opcfoundation.org/

[13] http://documentation.unified-automation.com/uamodeler/1.6.3/html/index.html

[14] https://github.com/brgl/libgpiod

[15] "CrossCompiling.pdf" - https://www.unified-automation.com/

[16] https://sourceforge.net/projects/raspberry-pi-cross-compilers