Sergey Morozov

Test Automation for Timing of Modbus Protocol Extension

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Bachelor’s Thesis

02 June 2023

Abstract

Author:	Sergey Morozov
Title:	Test Automation for Timing of Modbus Protocol Extension
Number of Pages:	25 pages + 4 appendices
Date:	02 June 2023
Degree:	Bachelor of Engineering
Degree Programme:	Electronics
Professional Major:	Electronics
Supervisors:	Panu Tuominen, Senior System Architect at Vaisala Oyj Janne Mäntykoski, Senior Lecturer at Metropolia UAS

A communication protocol used for data transmission between devices represents the key element of industrial systems. Modbus is one of the oldest industrial communication protocols that is still widely used in automation and control systems. Several versions of this protocol have been developed, from which the proprietary extension of Modbus RTU is used by Vaisala Oyj.

The aim of this work was to develop the effective approach for test automation of timing issues for the Vaisala extension of the Modbus protocol. Frame timing, timeouts and retries were examined using one of four combinations of tools selected during a pre-study phase: (i) a software Modbus simulator and sniffing software; (ii) a mixed domain oscilloscope; (iii) an FPGA board with a logic analyzer (iv); and a microcontroller with a logic analyzer. The last one was selected as the basis for the development of the test system.

The system was examined from both electrical signal and test automation perspectives. Two test cases were designed and executed on the test system as a proof-of-concept simulating the communication either with a client device or with a server one. The results of this work are proposed to be implemented to the current protocol tester used at Vaisala Oyj, contributing to the protocol quality assurance and the quality control of company products.

Keywords: Modbus RTU, protocol testing, system engineering, test automation

---

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

List of Abbreviations

1 Introduction 1

2 Theoretical Background 3

2.1 Overview of Modbus protocol 3

2.2 Physical Layer 4

2.3 Data Link Layer 5

2.4 Application Layer 6

2.5 Testing of Modbus Protocol 7

3 Materials and Methods 9

3.1 Pre-study 9

3.2 Hardware Setup 10

3.3 Software Architecture 12

3.4 Test Design 13

4 Results 14

4.1 Setup Implemetation 14

4.2 Test Automation Performance 16

5 Discussion 20

5.1 Benefits of Test Setup 20

5.2 Drawbacks and Potential Solutions 20

5.3 Further Perspectives 21

6 Conclusions 23

References 24

Appendices

Appendix 1: Procedure for software stack setup

Appendix 2: Automation software configurations for the logic analyzer

Appendix 3: Implementation code for Arduino Mega

Appendix 4: Test case implementation in pytest

List of Abbreviations

APU:	application protocol unit
AWG:	arbitrary waveform generator
CI:	continuous integration
CLI:	command line interface
DUT:	device under test
EIA:	Electronic Industries Alliance
EMC:	electromagnetic compatibility
GUI:	graphical user interface
FPGA:	field programmable gate array
PSU:	power supply unit
PLC:	programmable logical controller
RS:	recommended standard
RTU:	remote terminal unit
SCADA:	supervisory control and data acquisition
TCP:	Transmission Control Protocol
TIA:	Telecommunications Industries Association
UART:	universal asynchronous receiver-transmitter

1 Introduction

The unifying aim of this project is to determine the most effective approach for test automation of timing issues in a proprietary extension of the Modbus RTU (remote terminal unit) protocol developed at Vaisala Oyj, a Finnish company manufacturing industrial measurement equipment. Particularly, frame timing, timeouts and retries are examined using one of four combinations of tools selected during a pre-study phase: (i) a software Modbus simulator and sniffing software; (ii) an arbitrary waveform generator and an oscilloscope; (iii) an FPGA (field programmable gate array) board with a logic analyzer (iv); and a microcontroller with a logic analyzer. The first instrument in each pair of tools is responsible for the simulation or generation of Modbus messages on a bus, while the second one is used for sniffing and recording raw data of the reply from a device under test (i.e., a transmitter or a probe).

The following criteria are considered when comparing the results of four technical approaches mentioned above: latency, automation potential, test case development and maintenance speed, the total cost of the test setup, as well as the complexity of the test environment for an end-user. An optimal solution based on the pre-study analysis is a core of the protocol test system illustrated in Figure 1.

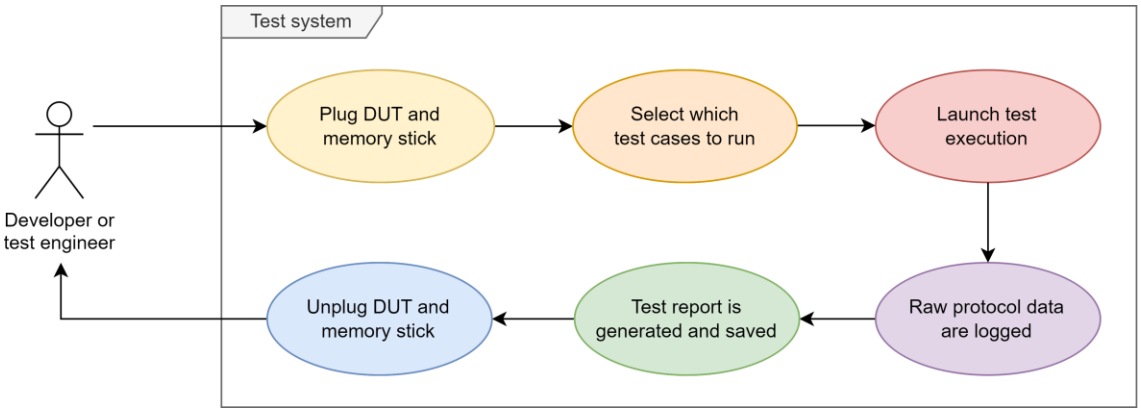

Figure 1. Conceptual diagram illustrating the scope and use case of the project. Note, the memory stick is an abstraction here, it can be replaced by a Cloud service or stored locally on a test PC.

The use case of the protocol test rack is that a developer or a tester comes to the laboratory and plugs in the device under test (DUT), selects test cases and launches their execution. DUT can be either a client device or a server device manufactured by Vaisala Oyj, also known as transmitters and probes, respectively. After a few minutes, the communication between DUT and the Modbus RTU simulator will be recorded, and a report will appear with percentage compatibility of DUT with the Vaisala Modbus extension protocol. Test cases for timing issues are defined according to the Modbus protocol specifications and proprietary Vaisala documentation and with a perspective of running in a continuous mode.

The results of this project, i.e., the selected technology and test cases, are proposed to be integrated into a larger test system designed for the proprietary Modbus extension at Vaisala. The use of the Vaisala extension is constantly growing among products because it enables autoconfiguration of Modbus addresses and other parameters, which standard Modbus over serial communication cannot do. As a result, a customer would benefit from the plug-and-play operation. Thus, the outcome of this work will directly contribute to the protocol quality assurance and the quality control of company products where this protocol extension is used.

2 Theoretical Background

2.1 Overview of Modbus Protocol

Modbus represents one of the oldest industrial communication protocols that is still prevalent in automation and control systems around the world. The protocol was developed by Modicon (later owned by Schneider Electric), the manufacturer of programmable logic controllers (PLC), and was officially published in 1979. Originally, Modbus interface was based on RS-232 serial communication (EIA/TIA-232 standard[1]), enabling master-slave (hereinafter client-server, respectively[2]) data transmission with distance limited to fifteen meters. At the device level, the client corresponds to a PLC, a microcontroller, or a supervisory computer, while the server can be either an input sensing device or an output load one. However, the following implementation of Modbus on RS-485 (EIA/TIA-485 standard), invented in 1984, allowed engineers to transform a point-to-point serial communication into a multi-drop network – an authentic fieldbus protocol. Henceforth, one client can control up to 32 servers using the same cable with the higher baud rate and the maximum length of 1200 meters. This qualitative improvement, coupled with parallel development of the Modbus protocol on Ethernet networks, explain its extensive application in the industrial sector. [1.]

The client-server architecture of Modbus defines that only a client device can initialize the communication and send commands called Modbus requests. In turn, every server device has a unique Modbus address and shall reply to a client request generating a Modbus message within a specified time interval. A

---

[1] Both EIA/TIA-232 and EIA/TIA-485 recommended standards will be referred as RS-232 and RS-485 (the most widely used nicknames for the standards), respectively, to simplify the text.

[2] According to Press Release of Modbus Organization published in 2020, master-slave is replaced with client-server as an occurrence of inappropriate language in official documentation.

client can send a request directly to a particular server referring to its Modbus address or broadcast to all available server devices. Using gateways and the Ethernet cable, clients can communicate with each other or with a supervisory computer forming a supervisory control and data acquisition (SCADA) system. [2; 3.]

Several versions of the Modbus protocol have been developed, among which two are widely used nowadays: Modbus RTU (remote terminal unit) and Modbus TCP (Transmission Control Protocol). Modbus RTU is implemented on RS-485 serial communication between PLC and field devices [2]. In contrast, Modbus TCP is used for routable communication over Ethernet connecting PLC with plant level devices or SCADA components. Technically, Modbus TCP represents an extra layer on top of Modbus RTU with differences in error checking and device addressing. [3.]

In addition, various extensions can be built on top of Modbus RTU suited for specific needs of companies or engineering communities [4; 5; 6]. Because of a proprietary Modbus protocol extension by Vaisala (hereinafter the Vaisala protocol) is based on Modbus RTU over RS-485, its physical, data link and application layers are reviewed in detail below from the perspective of the Vaisala protocol implementation (see also OSI model for layer descriptions).

2.2 Physical Layer

The physical layer of Modbus RTU is implemented over RS-485 using two-wire configuration, i.e., a twisted pair of positive and negative data lines (D+ and D-, respectively) to reduce electrical interference and provide half-duplex communication [2]. Common ground interconnects all the devices on a fieldbus. A client device shall have bias resistors (R_{bias}) on each data line, while a termination resistor (R_{term}) is optional. Usually, supply voltage for devices varies between 5 and 24 VDC and can be provided by external power supply unit (PSU) to each device on a field bus or by a client that can feed power directly to a server unless the client’s power limit is exceeded (Figure 2).

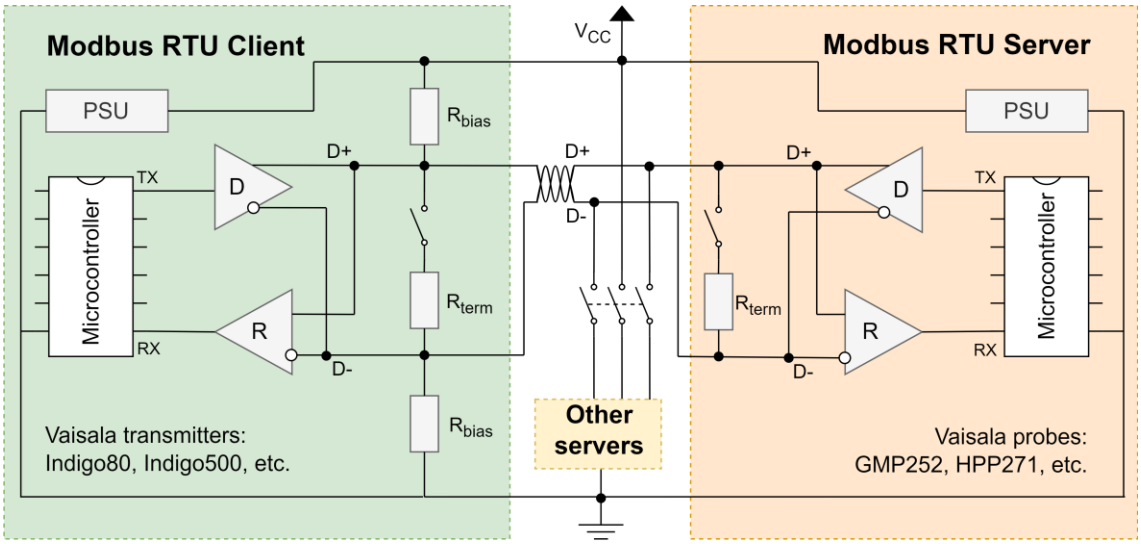

Figure 2. Principal diagram of half-duplex bus structure on RS-485 interface, where PSU is a power supply unit, TX and RX are transmitter and receiver pins of a microcontroller, respectively.

Either a point-to-point configuration or a multi-server one can be used between client and server devices (Figure 2). The cable length for the Vaisala protocol is recommended to be limited to 30 meters to follow Vaisala product specifications and electromagnetic compatibility EMC regulations (i.e., the IEC61326-1 standard). M8 and M12 connectors are suggested for the physical connection with clients and servers produced by Vaisala.

2.3 Data Link Layer

On the data link layer, a client device can broadcast Modbus messages to all the servers on a fieldbus. The broadcast mode shall be used only for writing commands on the server side, as namely, server devices shall not respond to the request. The reserved Modbus address for broadcast mode is 0. Alternatively, a client can send a request in unicast mode to an individual server with the unique Modbus address ranging from 1 to 247. According to the Vaisala protocol specifications, the default server address for the point-to-point configuration is 240. The server device is expected to respond without an unnecessary delay. In turn, a client does not have a unique address, because

only one Modbus RTU client can exist on the fieldbus and initialize communication with its servers. [2.]

Although serial port settings of Modbus RTU can vary, the Vaisala protocol requires fixed values for the baud rate and the serial format: 1 start bit, 19200 bits/s, 8 data bits, no parity bits, and 2 stop bits (on total one 11-bit asynchronous character). Each 8-bit data byte can be presented as two hexadecimal characters (e.g., 11111111_2 is FF_h), which form a Modbus RTU frame in a sequential manner. Silent intervals (≥ 3.5 characters) serve as separators between frames and its absolute time is dependent on the baud rate.

The timing requirements of Vaisala protocol are compliant with Modbus over serial lines specifications. However, additional timing requirements of the Vaisala protocol, such as timeouts and retries, are separately defined in its proprietary documentation.

Each Modbus RTU frame consists of four fields: the Modbus address (8 bits), the Modbus function code (8 bits), Modbus data (N × 8 bits), and error checking (16 bits). The last is also known as cyclic redundancy check (CRC), and it is calculated for each Modbus RTU message by both devices participating in protocol communication. If mismatch in CRC is observed or timing requirements are violated during data transmission, then no Modbus messages are generated on the application layer. [2.]

2.4 Application Layer

Modbus function code and Modbus data form the Modbus protocol data unit (Modbus PDU), which represents the core of a frame at the application layer. Services offered on the application layer are implemented in Modbus function codes which are defining behavior of Modbus client and server devices using the request/reply communication method. The functions are universal for all transmission modes, including Modbus RTU and Modbus TCP. In turn, Modbus data holds function subcode, arguments or data payload. Modbus PDU is a

central part of an application data unit (APU) that adds two high abstraction fields on a frame: the additional address and the error check corresponding to the Modbus address and CRC from the data link layer of Modbus RTU (Figure 3). [7.]

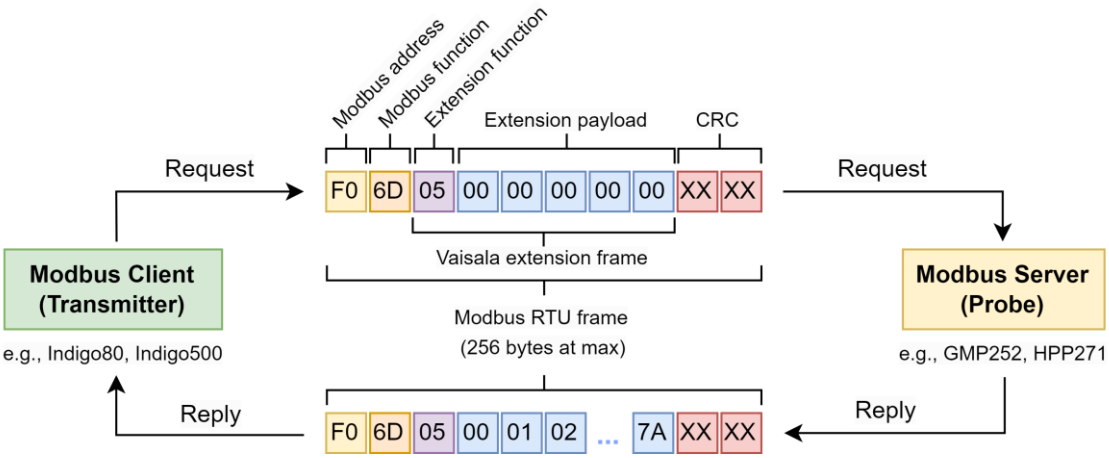

Figure 3. Serial communication between Modbus client and server devices. Vaisala extension frame is injected into Modbus RTU payload of request and reply to messages.

There are approximately twenty public functions designed for general purposes: reading and writing device registers, saving files, diagnostics including error messages, etc. In addition, around twenty other function codes are reserved for user defined functions. While the Vaisala protocol is built on the Modbus RTU frame (Figure 3), the usage of public Modbus functions is mostly optional. Instead, most operations of the Vaisala protocol have been implemented through one vendor specific function code $6D_h$ (109_{10}). Indeed, every Vaisala protocol function represents a subcode of the Modbus function $6D_h$ with the following payload marked as Extension function and Extension Payload, respectively, in Figure 3.

2.5 Testing of Modbus Protocol

Over the last 40 years of Modbus dominance in the industrial sector, different tools and techniques have been developed to test the protocol implementation.

Manual testing is still used on the physical and datalink levels of Modbus RTU, investigating protocol error detection and correction [8] and communication between PLC and Arduino prototyping board [9]. In this case, a test setup often includes electronic measuring instruments, such as an oscilloscope, signal generator, and power supply manipulator, which are typically difficult to automate [8; 10]. In contrast, the application layer of the Modbus protocol is often tested using software simulators sending Modbus messages to a fieldbus or a network [11; 12]. In addition, the security vulnerabilities have been intensively tested [13; 14].

Software simulators with a command line interface (CLI) or a graphical user interface (GUI) can pretend to be a client or a server. While GUI simulators, e.g., *Modbus Poll* and *Modbus Slave* (Witte Software®, Denmark), are easy to use, their automation potential is limited in comparison to CLI simulators (e.g., the Python package *pymodbus*). However, due to slow data processing of USB interfaces and an operating system, high latency can be an obstacle for testing of timing issues. In this case, an arbitrary waveform generator (AWG), an FPGA board, or a microcontroller can be a preferable solution to generate Modbus messages with minimal latency.

Test automation also enables continuous integration (CI) for a test setup creating long-term reports which simplifies regression testing and test case maintenance in general. To record raw data, software sniffers can be used (e.g., *socat* or the Python package *pyserial*). However, logic analyzers equipped with CLI automation software (e.g., *sigrok*) might be more optimal due to low latency and multichannel data capturing required for two-wire configurations of Modbus RTU. Finally, the recorded data, including timing and decoded bytes are used for test assertions of test cases. The selection of a test framework is less important and more dependent on user or company preferences.

3 Materials and Methods

3.1 Pre-study

To select the optimal solution for testing frame timing, timeouts and retries, several approaches had been reviewed at the pre-study phase. The ideas appeared during the R&D process and discussions with various specialists in the field are listed in Table 1. The pure software solution based on Modbus simulator and protocol sniffer had three serious drawbacks: high latency due to data transmission over the USB interface, inability to measure both lines simultaneously, and difficulty to find both client and server simulators inside one software package. Therefore, this option was rejected, even though it had been easy to automate and maintain.

Mixed domain oscilloscope represented a unique solution because it combined both arbitrary waveform generator and logic analyzer and can both generate Modbus replies or responses in addition to simultaneous data logging. While automation of a mixed domain oscilloscope work has been theoretically possible using Python and its *PyVISA-py* package, potential difficulties with triggering events based on incoming information coupled with excessive costs made this technical approach inefficient for implementation on test racks.

Table 1. Review of technical approaches for the protocol testing.

Technical approach	Costs (in €)	Advantages	Disadvantages
Modbus simulator & protocol sniffer	0	no hardware required easy to maintain	high latency (from 150 to 1500 µs)
Mixed domain oscilloscope	≥ 4000	low latency generate and read signals concurrently	expensive difficult to automate, particularly AWG
FPGA board & logic analyzer	≈ 1500	affordable low latency	hard to maintain required FPGA skills
Microcontroller with external clock & logic analyzer	≈ 1000	affordable easy to automate	latency is unclear required C/C++ skills

Two last ideas are rival because they have an affordable budget and could be automated because both an FPGA board and a microcontroller can be programmed for an immediate response to a byte trigger either as a client or as a server Modbus device (Table 1). However, after having a few trials with the FPGA board Diligent Artix-A7, it became clear that test case implementation on it might be extremally slow and a regular test engineer with basic programming skills might be not able to modify/fix a test case if necessary. Hence, from the maintenance point of view, this option was the most dangerous in a long term perspective. The microcontroller option, in turn, had unclear latency due to several minor factors including the quality of an external clock and the serial-RS-485 converter.

3.2 Hardware Setup

The actual test setup was based on the following core components: PC, Arduino Mega (Arduino, Somerville, US) with CMOS-RS-485 converter MAX13487 (Maxim integrated, San Jose, US), a logic analyzer Saleae Logic Pro 16, and the proprietary Vaisala relay board (Figure 4). PC was responsible for running test cases: setting up environment, making test assertions and tearing down. The logic analyzer is used to sniff and record the ongoing communication on Modbus RTU between DUT and Arduino Mega, which could act either as a client or as a server generating Modbus requests and replies, respectively. Only one DUT, either a probe or a transmitter, could be powered at once, which was controlled by the proprietary Vaisala relay board (Figure 4).

When the test case was initialized, PC sent commands over a USB cable to both Arduino Mega and the proprietary Vaisala relay board defining which DUT to power up. In turn, Arduino Mega compiled a new sketch assigned by the test case and either started sending requests as a client device to a Vaisala probe or waiting for a request from a Vaisala transmitter to reply as a server device.

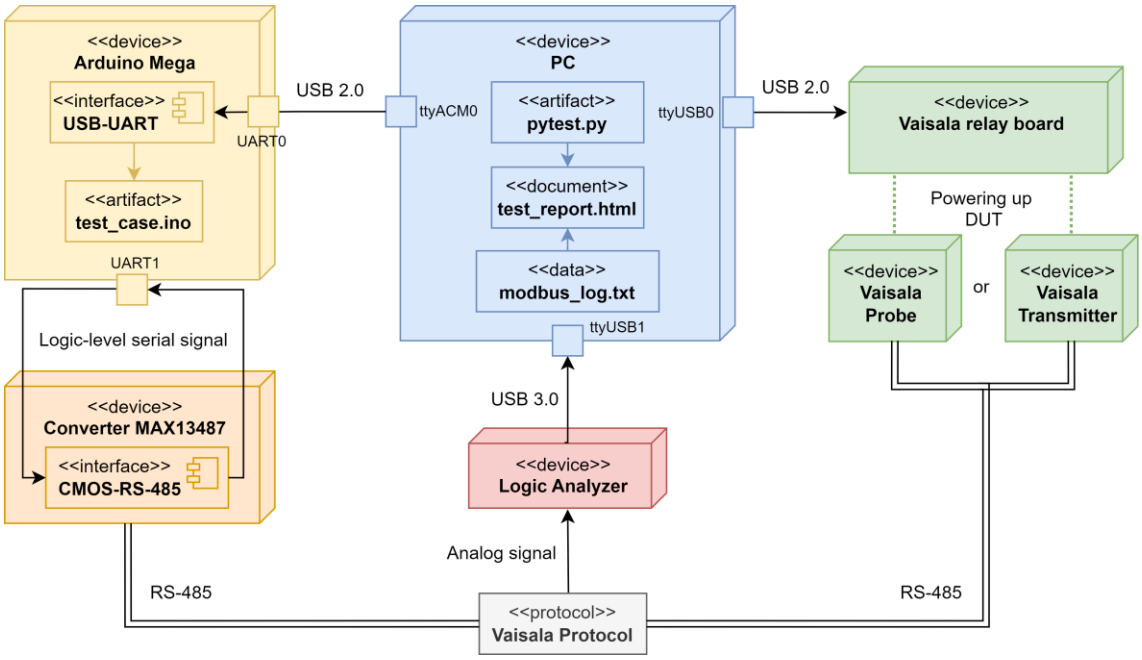

Figure 4. Deployment diagram of the protocol test setup.

Because another UART was needed to deliver the logic-level serial signal to the MAX13487 converter, the Arduino Mega board was selected for the test rack (the majority of Arduino boards have only one physical UART, while Arduino Mega has four). After signal conversion to Modbus RTU over RS-485, the logic analyzer started sniffing and recorded data transmission on both data lines between Arduino and Vaisala DUT. Saleae Logic Pro 16 was selected among other logic analyzers due to its cross-platform automation API and positive feedback reviews among the electrical engineering community [15].

The recorded data were sent back to PC and saved as a text file. The assertion was implemented on the PC using pytest framework comparing expected results with the actual data recorded by the logic analyzer. Finally, the tear down process was initialized to terminate communication between PC and other devices and performing software post-test execution actions.

3.3 Software Architecture

As the operating system of most end users is Linux Ubuntu (Canonical Ltd, London, UK), the whole software stack shall have Linux compatibility, while cross-platform software distributions were always considered as an advantage during the selection process. The other important requirement was the ability to use the test system software on a virtual machine because the environment setups of developers and test engineers are installed or preinstalled on VMware® Workstation (VMware Inc., Palo Alto, US). In addition, a command line interface (CLI) software was preferable for an end user in comparison to the software with a graphical user interface (GUI) due to the faster speed and better task automation capabilities on Linux using Shell scripting.

The VMware Workstation Pro software (version 17.0.1) was used to create a virtual machine on Linux Ubuntu (version 22.04 LTS) running on Windows PC. The environment setup was based on a series of shell commands in Linux Terminal to install all dependencies and modify configurations for the selected software. The *pytest* test framework, the logic analyzer automation API, and Vaisala relay board software were built and launched using the Python language, which was preinstalled on Linux distributions. As Python is the open source interpreted programming language, all above mentioned software could be fully controlled over the Terminal and Python scripts. The automation of Arduino Mega board was implemented in C/C++ programming language and Arduino CLI software to compile and upload Arduino sketch to the board. The generated Modbus RTU signal with or without intentional errors was defined in the Arduino sketch. The detailed setup instructions are demonstrated in Appendix 1.

The proprietary cross platform software Saleae Logic2 (Saleae Inc., San Francisco, US), version 2.4.7, was used to automate a logic analyzer. To run it in a headless mode (i.e., without showing GUI output), *Xvfb* (i.e., X11 virtual framebuffer) was launched. The automation API was based on the Python package *logic2-automation* allowing a user to write a script in Python

configuring the logic analyzer and the data capturing process. Digital voltage threshold was set to 1.65 V and sample rate 10 million samples per second controlled by a timer with one second measurement period. The asynchronous serial mode was selected for the decoding of electrical signals with the following configurations: 19200 bits/s, 8 bits per frame, 2 stop bits and no parity bits (see the Python code implementation in Appendix 2). In addition, the GUI version of Saleae Logic2 was used for the visual inspection of analog signals and debugging purposes.

3.4 Test design

Test cases were written to demonstrate operability and functionality of the system for testing timing issues. Particularly, serial port settings, frame timing and timeouts were checked using the small number of test cases as a proof-of-concept for the test setup. For each test case, either a client device (i.e., Vaisala Indigo500) or a server one (i.e., Vaisala GMP252) was communicating with the Modbus message generator (i.e., Arduino Mega with the RS-485 converter). The test cases were designed using the Gherkin style which offers better readability than a plain code and simplifies a review process of test case ideas (Listing 1).

```
Feature: [Test feature]
    Scenario [Test case]
    Given [Preconditions]
    When [Trigger event]
    Then [Expected output]
```

Listing 1. The structure of a Gherkin style test.

The test case implementation was based on plain Python using the *pytest* framework. To verify that test cases capture exit criteria, the results of test execution were also checked performing non-valid scenarios and using incorrect output in logs. The report for an end user was generated using the pytest plugin *pytest-html*. All the test cases were based on official specifications of the Vaisala protocol.

4 Results

4.1 Setup Implementation

All the hardware components of the system were mounted on the cardboard panel using M3 and M4 bolts and nuts. To enable easy access for connecting logic analyzer probes and 24 V power supply cable to RS-485 wires, the wire terminal was used. The use of T-splitters with M12 connectors allowed the test system to be physically minimalistic regarding wire connections (Figure 5).

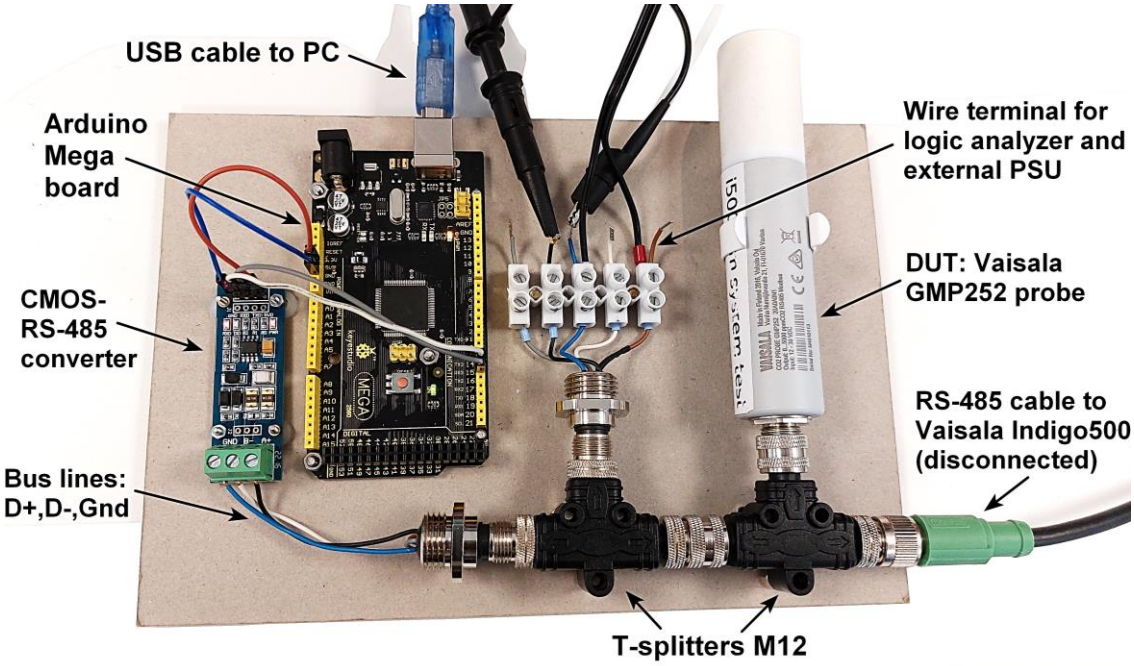

Figure 5. Hardware implementation of the test system.

The converter MAX13487 was powered from 3.3 V output pin of Arduino Mega instead of 5 V because of the bended analog signal form possibly related to the capacitance issues of MAX13487. Thus, the CMOS logic level serial signal was applied instead of the default TTL for that converter model. The rightmost T-splitter is used either to manually connect a probe, e.g., Vaisala GMP252, or a transmitter, e.g., Vaisala Indigo500 (Figure 5).

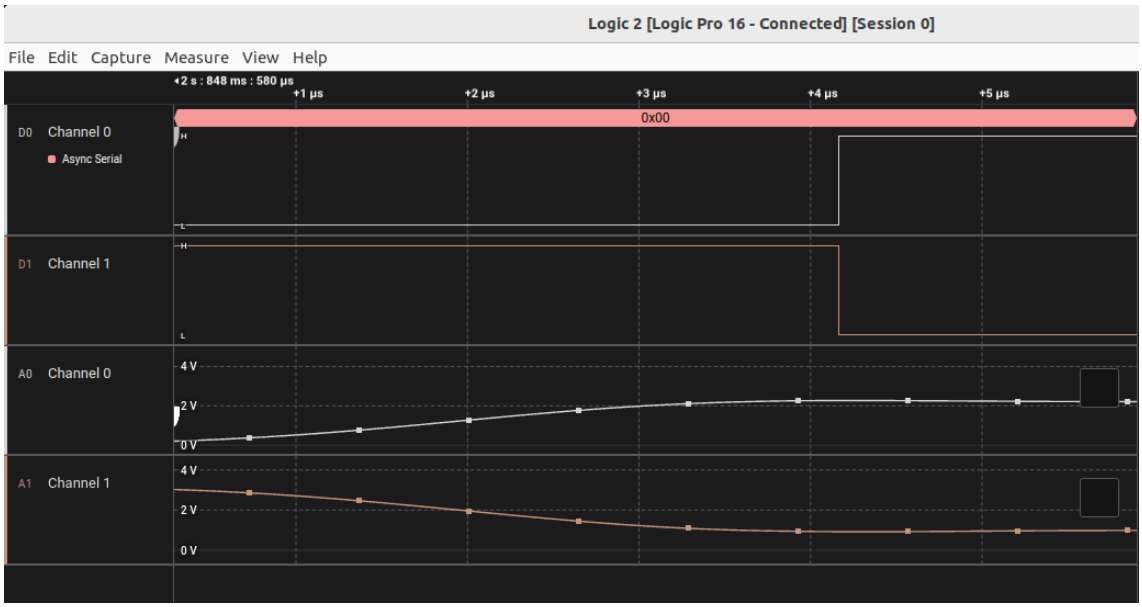

Figure 6. Analog and digital channels for D+ and D- data lines of RS-485 captured by a logic analyzer.

To evaluate the performance of Arduino Mega with MAX13487 as a Modbus message generator, timing differences in analog and digital channels on D+ and D- data lines of the RS-485 interface were inspected. Signals sent as a serial message from Arduino Mega and converted by MAX13487 from CMOS to RS-485 were analyzed using the GUI of Saleae Logic2. Regardless pure analog signal, the timing difference did not exceed 10 ns between D+ and D-. However, the variation in timing on digital level was between 62.5 to 250 ns, which is acceptable for the Modbus RTU testing. (Figure 6)

The digital voltage threshold set to 1.65V provided exactly the same decoded messages without seeing any mismatches and conflicts. However, when the threshold was lowered, the mismatches and errors appeared due the bended analog signal form on both data lines of RS-485 likely related to an unidentified capacity problem of the MAX13487 board (Figure 7). The time interval of one bit with digital zero was 52.32 µs, while one bit with digital one was 51.84 µs. However, the length of the entire 11-bit Modbus RTU character was fixed to 573 µs, which was according to the Modbus RTU specification for the baud rate of 19200 bit/s (Figure 7).

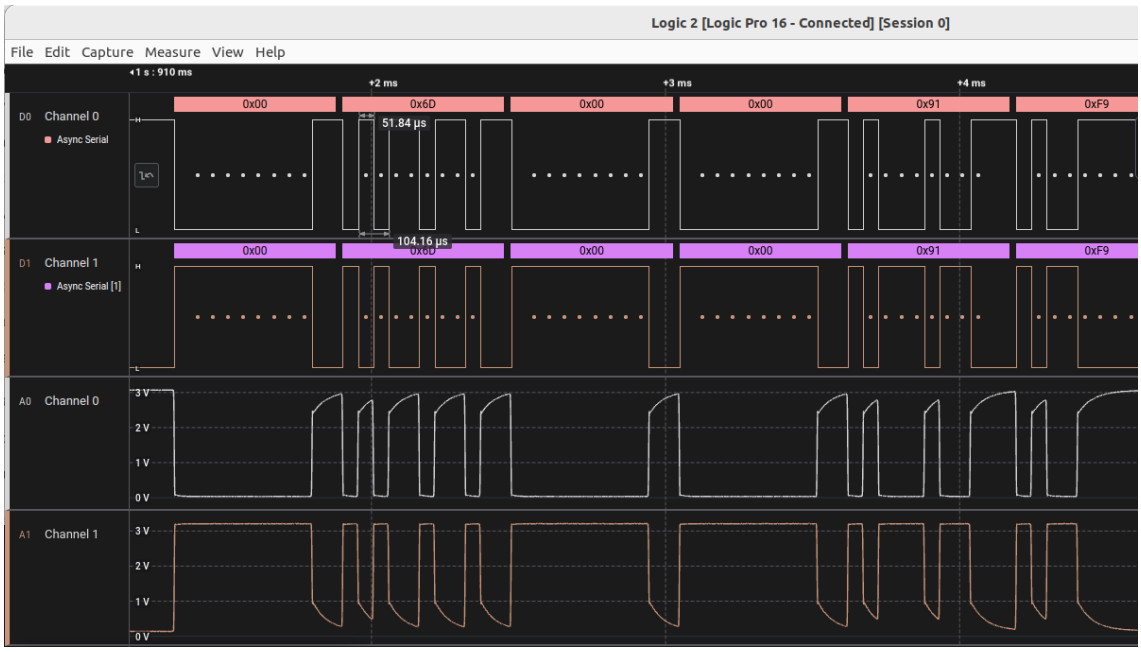

Figure 7. Analog and digital signals with decoded hexadecimal values forming one Vaisala protocol message generated by Arduino Mega.

The Modbus RTU messages generated by Arduino Mega and MAX13487 were recognized by both the Indigo500 transmitter and the GMP252 probe. The coded errors and timeouts in an Arduino sketch were generated without any issues with the same timing accuracy as mentioned above.

4.2 Test Automation Performance

Defining test cases in the Gherkin style was an effective approach for the discussion and reviewing them with colleagues. Two test cases were designed for the proof of concept: one for a client device and one for a server (Listing 2).

The implementation of test cases was based on assertions of the expected results with raw data captured by the logic analyzer. However, the preconditions of those test cases (see above) were different because different Vaisala protocol functions in the Arduino code had to be generated for different DUTs. Nevertheless, tearing down phase was realized in the same manner for all DUTs.

```
Feature: Test serial port format of client device

    Scenario: Client shall send Modbus request with 8N2 serial
              configurations
        Given logic analyzer is monitoring
        When client sends Modbus request in Modbus RTU mode with
             no parity
        Then Modbus request shall be in Modbus RTU mode And
                 one Modbus RTU byte contains 11 bits And
                 start bit of Modbus RTU byte shall be 0 And
                 two stop bits of Modbus RTU byte shall be 11

    Scenario: Server shall send Modbus request with baud rate
              19200 bps
        Given client is up and running
        When server is connected And
             logic analyzer is triggered
        Then time interval of Modbus RTU byte shall be 0.573 ms
```

Listing 2. Two test cases written in the Gherkin style for serial port settings.

The performance of automation software for the logic analyzer was relatively stable, especially when it was running in the headless mode, i.e., without GUI. However, USB concerns were observed once per several hours which were solved by the physical reconnection of a USB cable. The other problem was related to decoding: when 2 stop bits were selected in the protocol configuration window for serial asynchronous communication in the Logic2 software, the actual stop bit length was 0.5 bit less than expected. Thus, the post hoc correction in the test case implementation were required (see the comment about the formula in Figure 8).

The Python script was slightly modified from the example provided in the official documentation (see more details in Appendix 2). In turn, the Vaisala protocol function generator based on Arduino scripts and Arduino CLI was performing without any issues. Despite the simple approach of digital signal generation for these test cases (see Appendix 3), more complex use cases also demonstrated the high potential for further application of Arduino Mega in the setup development. The alternative approach of uploading all C/C++ code used by all test cases with index pointing of the required code was not tested.

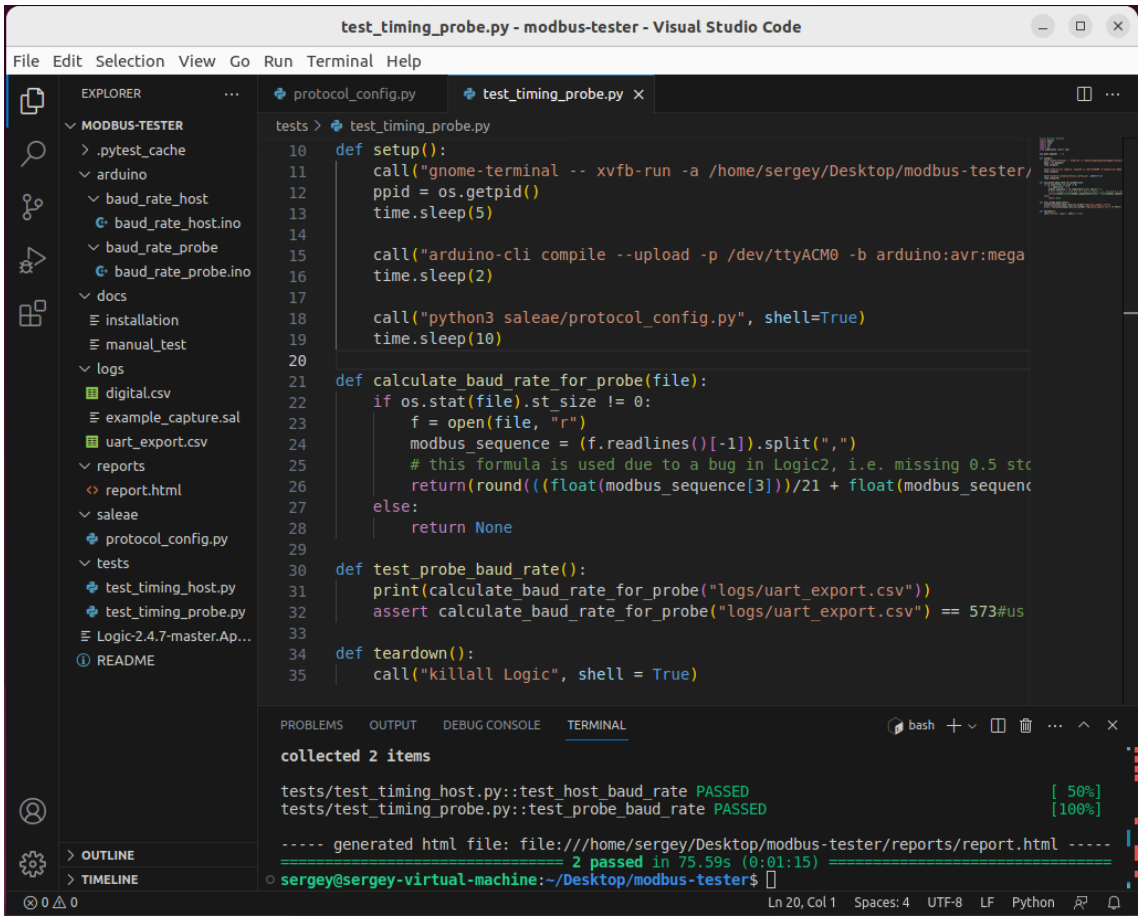

Figure 8. Two passed test cases where Arduino Mega represented as both a client and a slave device.

On average, the compiling and uploading Arduino code on the microcontroller took around 1.5 s, which was faster than it was initially expected. Thus, it was decided to use a separate Arduino sketch for each test case defining them in the precondition part using bash commands inside the Python script (Figure 8, Appendix 4).

Both DUTs passed tests successfully without variation in results. The accuracy of the Arduino-based generator of Vaisala functions was 120 ns, thus the determination of the baud rate was a trivial task for this setup.

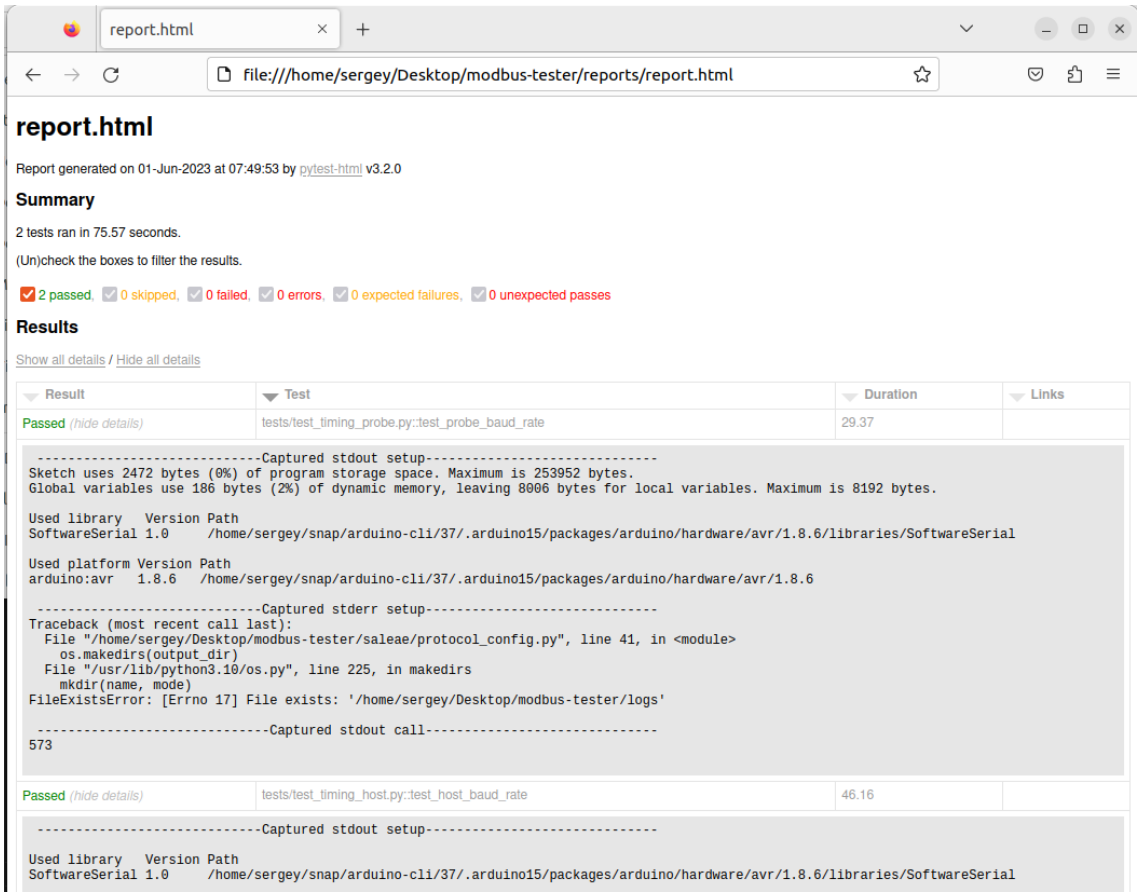

Figure 9. Test report based on *pytest-html*.

The test results were also presented in a form of the html report, which was quite useful for debugging purposes during the test implementation (Figure 9). In addition, it provides minimalistic statistics and the human-readable list of test cases which could be useful for the further state of the test system development, e.g., its presentation to end-users and stakeholders with limited coding skills.

5 Discussion

5.1 Benefits of Test Setup

The test system for testing of the Vaisala extension of the Modbus RTU protocol based on Arduino Mega and MAX13487 converter has several significant advantages comparing to other technical approaches mentioned in Table 1. First, all the components can be automated using the most popular programming languages, Python and C/C++, which makes this setup potentially used by many developers and test engineers in comparison to FPGA board. In addition, the Arduino CLI in combination with hardware peripherals of the Arduino Mega board significantly simplifies the compiling and uploading the C/C++ code from a bash or Python script to a microcontroller. Overall, it means that this protocol test system will be relatively easy to maintain and even integrate with other Vaisala projects because it is based on the similar software stack.

Second, the latency of Arduino Mega and CMOS-RS-485 converter was so insignificant in practice, that it will not impact the test procedure of Modbus RTU. Considering the complexity of FPGA board development or generating Modbus signals using a function generator in the mixed domain oscilloscope, the minimal lag of Arduino Mega can be neglected. Finally, the hardware setup is easy to install, debug, and customize. All the components can be replaced by alternative ones including a microcontroller board, a RS-485 converter, and even a logic analyzer. An affordable budget of the project allows replicating this setup and modifying it according to the application needs of company projects or teams.

5.2 Drawbacks and Potential Solutions

However, current test setup has a few limitations. The most noticeable one is the dependency on vendor-specific proprietary software Saleae Logic2. At the planning stage of this project, the open-source software Sigrok (sigrok.org) was

chosen to automate a logic analyzer in order to be vendor independent in a long-term perspective. Even though some logic analyzers performed well with Sigrok installed on virtual machines, Sigrok was not able to run with Logic Saleae 16 Pro. It might happen because Logic Saleae 16 firmware should be installed over USB every time when it is powered up which leads to errors on the Sigrok side. The open-source alternative to this software must be considered to avoid risks related to software maintenance or Saleae logic analyzer availability on the market.

In addition, Saleae Logic2 does not support decoding based on differential signal of RS-485 over a twisted-pair cable. Minor issues with the CMOS-RS-485 converter MAX13487 do not affect decoding process with Saleae Logic2. However, the problem with the bended analog signal form might occur if an alternative logic analyzer or automation software is used for this setup.

To avoid potential issues with the converter MAX13487, it can be replaced by the MAX485 or SN75176 transceiver. The logic analyzer Saleae Logic Pro 16 seems to be too advanced for the current application and can be substituted by more affordable models of the same brand: Logic Pro 8 or Logic 8. The development of a software extension for analysing differential signal of RS-485 would be more professional, however, time investments into this subproject should also be considered.

5.3 Future Perspectives

The pytest framework was approved to be suitable for the use case described here. Thus, more test cases should be written to increase the test coverage of the Vaisala protocol. A proper report system is necessary for an end user offering not only PASS/FAIL results, but also additional information related to timing, timeouts and retries issues. The compatibility metrics for DUT with the Vaisala protocol could be also informative for end users pointing out failed test cases and potential solutions for them.

The developed test system will be improved and integrated into the currently available application tester of the Vaisala protocol to be able to test the data link layer as well. The software stack could migrate from the current virtual machine to a Linux-based server and use Docker (Docker Inc., Palo Alto, US) container images to isolate the protocol tester with its dependencies into a self-contained unit.

6 Conclusions

The protocol tester for timing issues based on a microcontroller and a logic analyzer was successfully implemented demonstrating clear advantages over other technologies studied here. The distinct advantages of the selected technical approach are low latency of a generated digital signal, easy setup installation and maintenance, good potential for automation and integration into larger systems.

The proof-of-concept was shown based on two test cases simulating communication between the Vaisala protocol message generator with both a probe and a transmitter. The software stack including the test automation framework demonstrated high accuracy and stable performance.

The project offers a technological advancement for testing of the studied protocol which could be applied in the R&D department of Vaisala Oyj.

References

1 Cruceru A, Wüstrich L, Sattler P. 2022. Review of Industrial Control Systems Protocols. Network. pp. 27-31.

2 Modbus Organization. 2006. MODBUS Over Serial Line Specification & Implementation Guide: V1.02.

3 Modbus Organization. 2006. MODBUS Messaging on TCP/IP Implementation Guide: V1.0b.

4 Cena G, Cereia M, Bertolotti IC, Scanzio S. 2010. A Modbus extension for inexpensive distributed embedded systems. IEEE International Workshop on Factory Communication Systems Proceedings. pp. 251-260.

5 Brusell A. 2015 Modbus Interface Extension of D-flows Engineering Kit.

6 Ventuneac C, Gaitan VG. 2021. Implementation of an IIoT Access Gateway for the ModBusE–Modbus Extension using BeagleBone Black. EIRP Proceedings. 16(1).

7 Modbus Organization. 2012. MODBUS Application Protocol Specification: V1.1b3.

8 Urrea C, Morales C, Kern J. 2016. Implementation of error detection and correction in the Modbus-RTU serial protocol. International Journal of Critical Infrastructure Protection. 15: pp. 27-37.

9 Kuang Y. 2014 Communication between PLC and Arduino based on Modbus protocol. IEEE: Fourth international conference on instrumentation and measurement, computer, communication and control. pp. 370-373.

10 Lee TJ, Kang JW, Kim YH. 1999. An efficient automatic testing algorithm for power supplies in electronic systems. IEEE: Multimedia Technology for Asia-Pacific Information Infrastructure. 2: pp. 1446-1449

11 Voyiatzis AG, Katsigiannis K, Koubias S. 2015. A Modbus/Tcp fuzzer for testing internetworked industrial systems. IEEE: 20th conference on emerging technologies & factory automation. pp. 1-6.

12 Lecuivre J, Song YQ. 1995. A framework for validating distributed real time applications by performance evaluation of communication profiles. IEEE: International Workshop on Factory Communication Systems. pp. 37-46.

13 Jakaboczki G, Adamko E. 2015. Vulnerabilities of Modbus RTU Protocol – a case study. Nnals Of The Oradea University, Fascicle Of Management And Technological Engineering. pp. 203-206.

14 Morris TH, Gao W. Industrial control system cyber attacks. 2013. International Symposium for ICS & SCADA Cyber Security Research. pp. 22-29.

15 EEVblog. Electronics Community Forum [Internet]. 2021 [Cited April 12, 2023]. Available from: https://eevblog.com/forum/testgear/is-saleae-logic-analyzer-still-worth-it-nowadays/

Appendix 1. Procedure for software stack setup

1. Install and open VMware Workstation Pro, version >= 17.0.1.

2. Create a new virtual machine running on Ubuntu, version >= 22.04 LTS with: 50 GB SSD, 10 GB RAM, 2 processor cores, USB = 3.1, minimal setup, automatic updates.

3. In the Menu bar of VMware, click Edit >> Preferences >> USB and select "Connect the device to the foreground virtual machine".

4. Run the following commands in Terminal to install Logic2 software (version >= 2.4.7) in the project folder 'modbus-tester':

```
mkdir Desktop/modbus-tester/
cd Desktop /modbus-tester/
wget -O -x Logic2
"https://logic2api.saleae.com/download?os=linux"

chmod +x Logic2
sudo apt install libfuse2
gnome-terminal -- sh -x -c ./Logic2

cat /tmp/.mount_Logic2HKJ7kQ/resources/linux/99-
SaleaeLogic.rules | sudo tee /etc/udev/rules.d/99-
SaleaeLogic.rules > /dev/null && echo "finished installing
/etc/udev/rules.d/99-SaleaeLogic.rules"
```

5. Close the Logic2 software and connect the logic analyzer Saleae Logic Pro 16 to PC using a USB-3.0 cable.

6. Rerun Logic2 in Terminal. If your device is not connected, do the following troubleshooting:

a) open the .vmx file for the virtual machine using a text editor to remove the following line if it exists: usb.restrictions.defaultAllow = "FALSE";

b) in the Menu bar of VMware, click VM >> Settings >> USB Controller and check that USB compatibility is USB 3.1, not USB 2.0;

c) the logic analyzer is connected to PC using USB-3.0 cable, not USB 2.0;

d) repeat steps 3-6 and if the logic analyzer is still not working, contact us.

7. Install Arduino-CLI (version >= 0.32.2) in Terminal:

```
sudo snap install arduino-cli
sudo snap connect arduino-cli:raw-usb
arduino-cli core install arduino:avr
```

8. Connect Arduino Mega to PC and identify check the Arduino Mega board in Terminal:

```
arduino-cli board list
```

9. Create the Blink sketch using a text editor in Terminal:

```
arduino-cli sketch new Arduino_blink
nano Arduino_blink/Arduino_blink.ino

  void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
    Serial3.begin(19600, SERIAL_8N2);
  }
  void loop() {
    digitalWrite(LED_BUILTIN, HIGH);  delay(100);
    digitalWrite(LED_BUILTIN, LOW);  delay(100);
  }
```

10. Compile and upload the Blink sketch to a microcontroller of the Mega Board:

```
arduino-cli compile --upload -p /dev/ttyACM0 -b
arduino:avr:mega Arduino_blink
```

11. Install the pip and the Python package for Saleae LA automation:

```
sudo apt install python3-pip
pip install logic2-automation
```

Appendix 2. Automation software configurations for the logic analyzer

Note, this is a modified version of a sample code provided the manufacturer website www.saleae.com. Modified and new code lines are coloured in yellow and blue, respectively. The original comments in the code were omitted.

```
import os
import os.path
import sys
from datetime import datetime
from saleae import automation

sys.path.append('../')

with automation.Manager.connect(port=10430) as manager:
    device_configuration = automation.LogicDeviceConfiguration(
        enabled_digital_channels=[0, 1],
        digital_sample_rate=10_000_000,
        digital_threshold_volts=3.3)

    capture_configuration = automation.CaptureConfiguration(
    capture_mode =
         automation.TimedCaptureMode(duration_seconds=1.0))

    with manager.start_capture(
            device_configuration=device_configuration,
            capture_configuration=capture_configuration) as
               capture:
        capture.wait()

        uart_analyzer = capture.add_analyzer('Async Serial',
           label=f'Test Analyzer', settings={
              'Input Channel': 0,
              'Bit Rate (Bits/s)': 19200,
              'Bits per Frame': '8 Bits per Transfer (Standard)',
              'Stop Bits': 2,
              'Parity Bit': 'No Parity Bit (Standard)',
              'Significant Bit': 'Least Significant Bit Sent
                  First (Standard)',
              'Signal inversion': 'Non Inverted (Standard)',
              'Mode': 'Normal'
        })
        data_conf = automation.DataTableExportConfiguration(
            analyzer = uart_analyzer,
            radix = automation.RadixType.HEXADECIMAL)
```

```
output_dir = os.path.join(os.getcwd(), f'logs')
os.makedirs(output_dir)

analyzer_export_filepath = os.path.join(output_dir,
    'uart_export.csv')
capture.export_data_table(
    filepath=analyzer_export_filepath,
    analyzers=(uart_analyzer, data_conf))

capture.export_raw_data_csv(directory=output_dir,
    digital_channels=[0])

capture_filepath = os.path.join(output_dir,
    'example_capture.sal')
capture.save_capture(filepath=capture_filepath)
```

Appendix 3. Implementation code for Arduino Mega

Note, the real code for Vaisala protocol functions 2 and 3 are mocked.

```
#include <util/delay.h>
#include <SoftwareSerial.h>

const double inter_frame_gap = 6; // in ms (at min 2.005)

void setup(){
  Serial3.begin(19200, SERIAL_8N2);
}

void loop(){
  vaisala_protocol_function_1();
  delay(inter_frame_gap);
  vaisala_protocol_function_2();
  delay(inter_frame_gap);
  vaisala_protocol_function_3();
  delay(1000);
}

void Vaisala_protocol_function_1(void){
  // 00 6d 00 00 91 F9
  Serial3.write(0x00);
  Serial3.write(0x6d);
  Serial3.write(0x00);
  Serial3.write(0x00);
  Serial3.write(0x91);
  Serial3.write(0xf9);
}

void vaisala_protocol_function_2(void){
  // ...
}

void vaisala_protocol_function_3(void){
  // ...
}
```

Appendix 4. Test case implementation in pytest

Note, this example demonstrates communication between the Vaisala protocol function generator based on Arduino Mega and the Vaisala probe GMP252.

```
#!/usr/bin/env python3
import pytest
import time
import sys
import os
from subprocess import call

sys.path.append('../')

def setup():
    call("gnome-terminal -- xvfb-run -a
          /home/sergey/Desktop/modbus-tester/Logic-2.4.7-
          master.AppImage --automation", shell=True)
    ppid = os.getpid()
    time.sleep(5)

    call("arduino-cli compile --upload -p /dev/ttyACM0 -b
        arduino:avr:mega /home/sergey/Desktop/modbus-
        tester/arduino/baud_rate_probe", shell=True)
    time.sleep(2)

    call("python3 saleae/protocol_config.py", shell=True)
    time.sleep(10)

def calculate_baud_rate_for_probe(file):
    if os.stat(file).st_size != 0:
        f = open(file, "r")
        modbus_sequence = (f.readlines()[-1]).split(",")
        # this formula is used due to a bug in Logic2, i.e.
               missing 0.5 stop bit in timestamps
        return(round(((float(modbus_sequence[3]))/21 +
float(modbus_sequence[3]))*1000000))
    else:
        return None

def test_probe_baud_rate():
    print(calculate_baud_rate_for_probe("logs/uart_export.csv"))
    assert calculate_baud_rate_for_probe("logs/uart_export.csv")
        == 573#us

def teardown():
    call("killall Logic", shell = True)
```