Antero Vuorilehto

IoT Solution for Monitoring a Microbrewery

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Electronics

Thesis

17 September 2021

Abstract

Author(s):	Antero Vuorilehto
Title:	IoT Solution for Monitoring a Microbrewery
Number of Pages:	47 pages + 3 appendices
Date:	17 September 2021
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Electronics
Specialisation option:	Electronics
Instructor(s):	Anssi Ikonen, Senior Lecturer
	Erkki Räsänen, Technical Director

The goal of this thesis project was to implement an IoT solution for monitoring a microbrewery in Metropolia University of Applied Sciences and to write a full user manual on how to use and configure the solution according to a future user's needs. The making of the solution continued an innovation project first started by students of Metropolia University of Applied Sciences.

The finished project allows future users of the microbrewery to easily follow the values of a Brix-meter of the microbrewery, the temperature and pH-value of the liquid in the brewing kettles and even the states of the brewing process. Brix-value indicates the amount of sucrose in 100 grams of liquid. Monitoring the Brix-value is critical for the brewing process because it allows brewers to figure out the alcohol content of the finished product.

During the thesis project, the serial communication interface and the communication protocol used by the microbrewery and the possible hardware equipment and software platforms to be used in the finished project were studied. Based on the study, a theoretical research on the serial communication interface and communication protocol, as well as presentations of the equipment and platforms used in the project are included in the thesis.

The benefits of this thesis and its project is to help future users of the microbrewery in Metropolia University of Applied Sciences to remotely access the monitoring data of the microbrewery. The full user manual also gives the means for a future user to improve the features of the project for further development.

Keywords: RS485, Modbus RTU, microbrewery, electronics, Controllino, Arduino, c-programming

Tiivistelmä

Tekijä(t):	Antero Vuorilehto
Otsikko:	IoT-ratkaisu pienpanimon valvontaan
Sivumäärä:	47 sivua + 3 liitettä
Aika:	17.9.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Sähkö- ja Automaatiotekniikka
Suuntautumisvaihtoehto:	Elektroniikka
Ohjaaja(t):	Tutkintovastaava Anssi Ikonen Tekninen ohjaaja Erkki Räsänen

---

Opinnäytetyön tavoitteena oli toteuttaa IoT-ratkaisu pienpanimon valvonnan avuksi sekä kirjoittaa käyttöopas kehitetyn IoT-ratkaisun käyttöä ja konfigurointia varten. Pienpanimo sijaitsee Metropolia Ammattikorkeakoulussa. Työ jatkoi innovaatioprojektia, jonka Metropolian opiskelijat ovat aikaisemmin aloittaneet.

Insinöörityössä kehitetyn IoT-ratkaisun avulla tuleva pienpanimon käyttäjä voi etäyhteyden avulla valvoa pienpanimon käymisprosessin aikana tarvittavia tietoja, kuten Brix-mittarin arvoja, käymissäiliöissä olevien nesteiden lämpötiloja ja pH-arvoja sekä käymisprosessin eri vaiheita. Työn tärkeimpänä tavoitteena oli luoda Brix-arvosta reaaliaikainen kaavio, jota voitaisiin seurata etäyhteydellä. Brix-arvo indikoi sakkaroosin määrää 100 grammassa nestettä, joka on erityisen tärkeää oluen panemisen kannalta. Seuraamalla Brix-arvoa, voi panimon käyttäjä arvioida valmiin tuotteen alkoholipitoisuutta.

Insinöörityöprosessin aikana tutustuttiin pienpanimon käyttämään sarjaliikennerajapintaan ja kommunikointiprotokollaan sekä mahdollisiin laitteistoihin ja alustoihin, joita voitaisiin käyttää valmiissa työssä. Tutustumisen pohjalta luotiin teoreettinen tutkielma pienpanimon sarjaliikennerajapinnasta ja kommunikointiprotokollasta sekä esittelyt työssä käytetyistä laitteistoista ja alustoista.

Tämän opinnäytetyön ja sen projektin hyötynä on auttaa tulevia Metropolia Ammattikorkeakoulun pienpanimon käyttäjiä pääsemään helposti käsiksi sen valvontaan liittyviin tietoihin etäyhteydellä. Täydellinen käyttöopas antaa ohjeet, kuinka jatkokehittää ja räätälöidä projektin ominaisuuksia tulevan käyttäjän tarpeiden mukaan.

Avainsanat: RS485, Modbus RTU, pienpanimo, elektroniikka, Controllino, Arduino, c-ohjelmointi

Contents

List of Abbreviations

ASCII	American Standard Code for Information Interchange
CRC	Cyclic redundancy check
CR LF	Carriage Return-Line Feed
COM	Communication Port
DIP	Dual In-line Package
DIY	Do It Yourself
GND	Signal Ground
IDE	Integrated Development Environment
IoT	Internet of Things
LRC	Longitudinal Redundancy Check
LSB	Least Significant Bit
MSB	Most Significant Bit
PLC	Programmable Logic Controller
PWM	Pulse-Width Modulation
RTU	Remote Terminal Unit
SCI	Serial Communication Interface
SPI	Serial Peripheral Interface

TCP/IP	Transmission Control Protocol / Internet Protocol
TTL	Transistor Transistor Logic
XOR	Logical Exclusive OR

1 Introduction

The thesis project was made for Metropolia University of Applied Sciences. The aim for the project was to create an IoT solution for a microbrewery, custom made for the university by Tankki oy, Finland during winter 2015 and spring 2016. A microbrewery is a brewery that produces small amounts of beer. The microbrewery was acquired by Metropolia University of Applied Sciences for research and educational purposes.

The thesis consists of a theoretical background part where the theory behind the serial interface (RS485) and the communication protocol (Modbus RTU) used in the project is explained. After that the hardware and software platforms used in the thesis are introduced and explained. The finished project and its results are shown and described in the fifth chapter. The final chapter is the conclusion of the thesis.

The first goal of the thesis project was that the value from a Brix meter inside the microbrewery is logged and graphed into cloud so the user can read it from their computer. The second goal was to write a user manual on how to configure the user interface and the values sent to the IoT platform. The manual is included in Appendix 1.

2 Theoretical Background

The theory behind the serial interface and the communication protocol used in the thesis is extensive. This chapter will clarify the theory in detail.

2.1 RS485

ANSI TIA/EIA-485, commonly known as RS485 is a serial communication interface (SCI) that was created in 1998. It is widely used in data acquisition and control applications where multiple nodes communicate with each other. It is used in industrial applications as well as hobby projects. [1.]

RS485 network consists of a single pair of wire and a ground wire, to have up to 32 devices at 1200 meters distance to communicate at half-duplex to each other. The length of the network can be extended by adding RS485 repeaters every 1200 meters and the number of devices used in the network can also be increased by 32 devices with per one repeater. Since every RS485 device needs to have a unique address, the maximum number of devices in a network with repeaters is typically 256. [2.]

The standard defines the electrical characteristics for the drivers and receivers used in serial communication, the physical layer. In a “two-wire” configuration, RS485 uses two balanced and differential signal lines called “A” and “B”. A balanced signal line is typically a twisted pair cable with matching impedances. In RS485 the transmitter and receiver must be impedance matched as well. This chapter will focus on the “two-wire” configuration.

A common RS485 topology is twisted pair cable between the RS485 transceivers of each device and terminator resistors at the first and the last transceiver. The termination resistors are used to ensure signal integrity and avoid reflections in the transmission lines. Figure 1 is an example of this kind of

topology. In some cases the termination requirements and the device arrangements may vary.

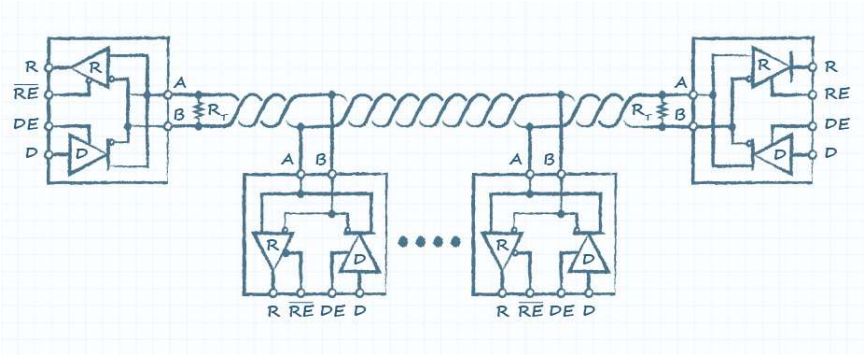


Figure 1. A typical RS485 multi-drop network [3]

In an RS485 interface signals A and B are a differential pair, where one of the wires carries the original signal and the other one carries an inverted version of the original signal. Differential signals are superior to single-ended interfaces especially with long distances.

Voltages tend to drop when distances get longer, this harms the signal integrity because a single-ended interface receiver references the original signal to the ground. With a differential receiver, the signals are referenced to each other, in other words the receiver looks for the voltage difference of the differential pair. Then it reconstructs the pair of signals back into one signal, which the host device can read. This allows for better signal integrity over long distances.

Noise and electrical interference may also affect a cabling in a system, especially when the cables get long. A balanced twisted pair cable and a differential receiver will combat this very well. When there is interference or noise spike on one of the signals, it will also affect the other signal, and because the differential receiver references the signals to each other, this cancels the effect of noise or interference on the signal. This ability is also known as common mode rejection. Figure 2 shows a visual representation of what happens to a differential signal with common mode rejection.

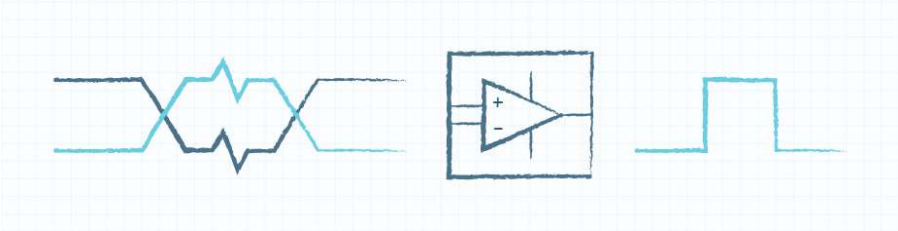


Figure 2 Common mode rejection in action [3]

RS485 bus does not require a specific voltage to transmit data, but it requires the differential voltage between “A” and “B” to be at least +/-200 mV at the receiver. Standard RS485 transceivers allow a common-mode voltage range of -7 V to +12 V, this allows an RS485 bus to have devices which transmit different voltages, if it is in the range. This is also important with longer cables because the transmitting device can transmit a higher differential voltage, and even though it drops while traveling the cable, as long as it does not drop below +/- 200 mV, the signal will be fine. [3.]

Figure 3 shows a comparison of different physical layer protocols and as we can see, RS485 is the fastest and most versatile there.

SPECIFICATIONS		RS-232	RS-423	RS-422	RS-485
Mode of Operation		SINGLE -ENDED	SINGLE -ENDED	DIFFER- ENTIAL	DIFFER- ENTIAL
Total Number of Drivers and Receivers on One Line		1 DRIVER 1 RECVR	1 DRIVER 10 RECVR	1 DRIVER 10 RECVR	1 DRIVER 32 RECVR
Maximum Cable Length		50 FT.	4000 FT.	4000 FT.	4000 FT.
Maximum Data Rate		460kb/s	100kb/s	10Mb/s	30Mb/s
Maximum Driver Output Voltage		+/-25V	+/-6V	-0.25V to +6V	-7V to +12V
Driver Output Signal Level (Loaded Min.)	Loaded	+/-5V to +/-15V	+/-3.6V	+/-2.0V	+/-1.5V
Driver Output Signal Level (Unloaded Max)	Unloaded	+/-25V	+/-6V	+/-6V	+/-6V
Driver Load Impedance (Ohms)		3k to 7k	>=450	100	54
Max. Driver Current in High Z State	Power On	N/A	N/A	N/A	+/-100uA
Max. Driver Current in High Z State	Power Off	+/-6mA @ +/-2v	+/-100uA	+/-100uA	+/-100uA
Slew Rate (Max.)		30V/uS	Adjustable	N/A	N/A
Receiver Input Voltage Range		+/-15V	+/-12V	-10V to +10V	-7V to +12V
Receiver Input Sensitivity		+/-3V	+/-200mV	+/-200mV	+/-200mV
Receiver Input Resistance (Ohms)		3k to 7k	4k min.	4k min.	>=12k

Figure 3. Comparison between different physical layer protocols [4]

2.2 Modbus Serial

Modbus is one of the first widely used fieldbus. It is an open serial communication protocol used in industrial networks, originally published in 1979 by Modicon for their own PLCs. Modicon is nowadays a branch of Schneider Electric's. Modbus was originally only a serial communication protocol, but it has expanded to TCP/IP as well.

Modbus RTU and Modbus ASCII are a request-response protocol with a master-slave relationship, in other words it has one master that controls the data transactions with one or several slaves that respond to the masters'

requests. Figure 4 is a basic example of Modbus serial architecture representing the master-slave relationship.

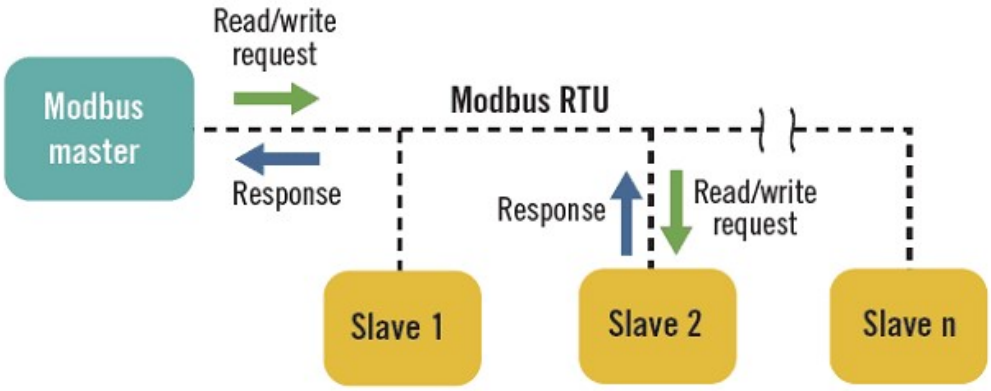


Figure 4. Modbus serial architecture [5]

A newer variant to Modbus family, Modbus TCP uses client/server architecture as seen in Figure 5. It was created to allow Modbus RTU/ASCII protocols to be carried over ethernet. [5.]

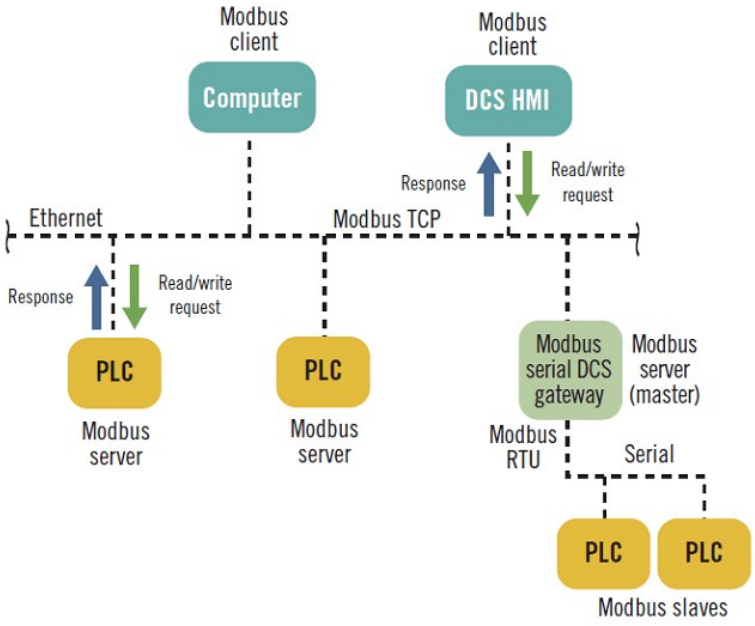


Figure 5. Modbus TCP architecture [5]

2.2.1 Modbus Serial Framing

The communication sequence between Modbus master and slaves begins by the master sending a request or a command on to the bus for the slaves. This is called “a query”. The slaves will then act according to the command received, supply the data requested or reply with an error if the command or request ca not be carried out. In case of an error, the query will be ignored. In no circumstances will the slaves transmit data on the bus unless required to do so by the master.

A slave will return “a response” to the master if it was specifically addressed in the query. The response will supply the requested data, confirm that the message was received, or reply with an error. A Modbus master can also “broadcast” to all slaves. In that case the slaves will not give a response to the master. [6.]

Standard Modbus network controllers can be configured as ASCII or RTU transmission mode, but they must be configured to only one. Modbus RTU and ASCII transmission modes are not compatible with each other because their frames are different.

Table 1. Modbus ASCII frame [7]

Start	Address	Function	Data	LRC	End
:	2 Chars	2 Chars	N Chars	2 Chars	CR LF

When looking at Table 1, in Modbus ASCII configuration each eight-bit byte in a message is two ASCII-characters. Data transmission is slower than in RTU configuration, but it allows up to one second time interval between characters without causing an error. The message starts with a ‘:’ character and ends with CR LF (carriage return-line feed).

Coding system:	Hexadecimal, ASCII characters: 0-9, A-F Two ASCII characters per 8 bit data
Bits per byte:	1 start bit, 7 data bits (least significant bit sent first) 1 bit for even/odd parity (no bit for no parity) and 1 or 2 stop bits
Error check field:	Longitudinal redundancy check (LRC)

Figure 6. Modbus ASCII format for each byte [6]

Slaves in the bus are continuously polling for ':' character, if the character is received, the slaves will then check the address field to find out if the message was intended to it. Next in the frames are the function field and the data field, which will be explained in depth a bit further in the thesis. Before the end CR LF there is error checking field. The error check characters are the result of LRC calculation which is performed on the message contents. Each byte in Modbus ASCII follows the format represented in Figure 6.

Modbus RTU frames are different from Modbus ASCII frames. Each eight-bit byte in a message is transmitted by two four-bit hexadecimal characters. This allows for greater character density and greater data rates with same baud rates than in ASCII configuration. While ASCII configuration allowed for up to one second time interval between characters, RTU messages must be sent continuously. Modbus RTU frame is shown in Table 2.

Table 2. Modbus RTU frame [7]

Start	Address	Function	Data	CRC	End
3.5 Char time	8 Bit	8 Bit	N * 8Bit	16 Bit	3.5 Char time

While in ASCII mode the frame starts with ':' character, Modbus RTU frame begins with 3.5-character times gap in transmission. If there is no transmission on the bus for 3.5-character times, the devices on the network start to search for their address, which is sent with two four-bit hexadecimal characters. Each device in the network will decode the address field. After address, the frame

continues with Function code, data, and the CRC frame. If the CRC sent does not match the CRC calculated on the receiving side, the message will be ignored, and the slave will send an error to the master. The frame ends after another 3.5-character times interval.

If there is longer than 1.5-character times interval of silence in the transmission, the whole message will be deemed incomplete, and the receiving device returns to wait for the next address field. If a new message begins before the 3.5-character times in the end, the device will think that the message is not complete and thinks it is continuation of the previous message. The receiving device will then set an error since the CRC value will not be valid for the combined messages. Each byte in Modbus RTU follows the format shown in Figure 7. [7.]

Coding system:	8-bit binary, comprised of two 4-bit words Each word equivalent to one hexadecimal character
Bits per byte:	1 start bit, 8 data bits, least significant bit sent first, 1 bit for even/odd parity; no bit for no parity and 1 or 2 stop bits
Error check field:	Cyclical Redundancy Check (CRC)

Figure 7. Modbus RTU format for each byte [6]

How long the 3.5-character times is will depend on the baud rate. If the baud rate is higher than 19200, Modbus states that 1.5-character times must be fixed to 750 microseconds and for 3.5-character times 1750 microseconds. For baud rates under 19200 the character times must be calculated. Let's use 9600 baud rate as an example:

Calculating the Modbus RTU character time (1)

$$\frac{11\frac{bits}{char}}{9600\frac{bits}{s}} = 0.001145s = 1.145ms$$

With 9600 baud rate, 1-character times is 1.145ms as seen in Equation 1. So, 1.5-character times is 1.715ms and 3.5-character times is 4.0075ms as.

Table 3 is an example to show the difference between one Modbus ASCII and Modbus RTU message:

Table 3. Difference between Modbus RTU and Modbus ASCII message

	Message	RTU	ASCII (hex)
Start	-	3.5-character times	3A (1 byte)
Address (slave)	30 (slave 30)	1E (1 byte)	33 30 (2 bytes)
Function	03 (read holding registers)	03 (1 byte)	30 33 (2 bytes)
Starting address high		00 (1 byte)	30 30 (2 bytes)
Starting address low	99 + (40001 offset)	63 (1 byte)	36 33 (2 bytes)
Number of registers high		00 (1 byte)	30 30 (2 bytes)
Number of registers low	3	03 (1 byte)	30 33 (2 bytes)
Error check	-	F7 BA (2 bytes) CRC calculator	37 39 (2 bytes) LRC calculation (79 in hex)
Stop	-	-	0D 0A = CR LF (2 bytes)
Total bytes		8	17

This message will read holding registers with Function code 03, from slave number 30, starting from 40100 to 40102. As we can see Modbus ASCII needs a lot more bytes to accomplish the same outcome.

2.2.2 Modbus Address Field

All the fields in a Modbus frame have a specific job to do. The next sub-chapters will go through each of the fields more in depth.

After the message starts with ':' character in ASCII configuration or 3.5-character times of silence in RTU configuration, all the devices in the network will start to look for their own address which is sent in the first field of Modbus message, the address field.

Simply put, Modbus master will address a specific device in the address field with one byte in RTU configuration, or two bytes, in other words two characters in ASCII configuration. When a slave responds to the master, it will put its own address in the address field of the response. Valid addresses range from 1 to 247 and address 0 is reserved for broadcasting to all slaves in the network. Slaves do not respond to broadcast messages.

2.2.3 Modbus Function Field

The function field's purpose is for the master to tell the slave what to do. Modbus master will once again send one byte in RTU configuration and 2 characters in ASCII configuration inside the function field of the Modbus message frame. The function code can be from 1 to 255 but most of the function codes are either reserved for future uses, not in use or not implemented in a module.

Table 4. Common function code for Modbus

Function code	What it does
01 (01 RTU / 30 31 ASCII)	**Read coil status** 0x reference
02 (02 RTU / 30 32 ASCII)	**Read input status** 1x reference
03 (03 RTU / 30 33 ASCII)	**Read holding registers** 4x reference
04 (04 RTU / 30 34 ASCII)	**Read input registers** 3x reference
05 (05 RTU / 30 35 ASCII)	**Write single coil** 0x reference
06 (06 RTU / 30 36 ASCII)	**Write single register** 4x reference
15 (0F RTU / 31 35 ASCII)	**Write multiple coils** 0x reference
16 (10 RTU / 31 36 ASCII)	**Write multiple registers** 4x reference

Table 4 shows some of the more common function codes and what the function code does. Coils are 1-bit registers which can be read or written. Discrete inputs are 1-bit registers for read only functions. With coils you can read or write the state of a switch but with discrete inputs you can only read the state. Holding registers and input registers are both 16-bit registers. Holding registers are probably the most used register in Modbus, it can be read or written. Input registers are read only. The references written in the “what it does” field of the Table 4 are used to reference to the right register:

- Coils (0x) = 00001 – 09999
- Discrete inputs (1x) = 10001 – 19999
- Input registers (3x) = 30001 – 39999
- Holding registers (4x) = 40001 - 49999

Every function code has a unique query and response. To understand how they work, there will be some examples. The examples will be addressing imaginary

slave 30 in the network. The error checking field will be left empty, and the queries and responses will be shown in Modbus RTU for simplicity.

Function code 01, read coil status:

- Reads the ON/OFF status of a discrete coil
- Specifies the starting coil and the number of coils to be read
- Logic 1 is ON and logic 0 is OFF
- In the response one coil corresponds to one bit of data
- First coil corresponds to the LSB of the response

Table 5 shows an example of Function code 01 query and Table 6 shows the response for the query. A query is sent to read the states of coils 0, 1 and 2. Coils 0 and 2 are conducting current, so the state of those will be logic 1. Coil 1 is not conducting current, so its state will be logic 0.

Table 5. Function code 01 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	1	01
Starting address high	0	00
Starting address low	0	00
Number of coils high	0	00
Number of coils low	3	03
Error checking	-	-

Table 6. Function code 01 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	1	01
Byte count	1	01
Data	5	05
Error check	-	-

A query was sent to read the state of coils 0 to 2. In the response we get the byte count which in this case is one. If we would have read coils from 0 to 8, we would have gotten two bytes of data because one coil corresponds to one bit of data and Modbus messages have eight bits reserved for data, so one byte.

The content of the data field of the response is decimal five, hexadecimal 05 which in binary equals to 0000 0101. Since the first coil read corresponds to the LSB of the data byte and then next coil is the next bit and so on. It can be seen that coil 0 conducts current, coil 1 does not conduct current and coil 2 conducts current.

Function code 02, read input status:

- Reads the ON/OFF state of a discrete input register
- Specifies the starting coil and the number of coils to be read
- Logic 1 is ON and logic 0 is OFF
- In the response one discrete input register corresponds to one bit of data
- First discrete input register corresponds to the LSB of the response

Function code 02 is almost the same as Function code 01, but the reference code is 1x, it references to different registers, the discrete 1-bit input registers. Tables 7 and 8 show the query and response of Function code 02. In the query, the discrete input registers 0 to 10 are read, meaning discrete input registers 10001 to 10011. The reference 1x is integrated into the function code, in other words, when reading discrete input register 0 with Function code 02, it will be referenced to the discrete input register 10001.

Table 7. Function code 02 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	2	02
Starting address high	0	00
Starting address low	0	00
Number of inputs high	0	00
Number of inputs low	11	0B
Error checking	-	-

Table 8. Function code 02 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	1	01
Byte count	2	02
Data (inputs 7 to 0)	177	B1
Data (inputs 10 to 8)	3	03
Error check	-	-

Since the query is sent to read more than eight input registers, the response will contain two bytes of data. First byte will contain the data starting from the first discrete input register read up to the eighth and the second byte of data will contain the data from discrete input registers 9 to 10.

The data field of the response consists of B1 03 in hexadecimal, which in decimal is 177 and 3 and in binary 10110001 and 00000011. Table 9 and 10 show which discrete inputs according to the response data are on and off:

Table 9. Function code 02 example responses first data byte

1	0	1	1	0	0	0	1
Input 7	Input 6	Input 5	Input 4	Input 3	Input 2	Input 1	Input 0
ON	OFF	ON	ON	OFF	OFF	OFF	ON

Table 10. Function code 02 example responses second data byte

0	0	0	0	0	0	1	1
-	-	-	-	-	Input 10	Input 9	Input 8
-	-	-	-	-	OFF	ON	ON

Function code 03, Read holding register:

- Reads the content of 16-bit holding registers
- Specifies the starting holding register and the number of holding register to read
- In the response one holding register corresponds to two bytes of data
- First byte contains the high order bits, and second byte contains the low order bits

Holding registers are read or write registers which usually are used to set an analog output, most commonly 0-10V or 4-20mA, which control a thermostat or a fan. Nowadays there are many applications where holding registers are used also as analog input registers. The reference of holding registers is 4x which is implemented in the function code. Holding register 1 is addressed as 0 and is referenced to the holding register 40001 in this case.

Let's say a query is sent to read the holding register 4. It contains data which is used to set an analog output of 0-10V to 5V. A 16-bit Digital to analog converter is used, so 0V would be 0, 10V would be 65536. The query and response of Function code 03 is shown in Tables 11 and 12.

Table 11. Function code 03 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	3	03
Starting address high	0	00
Starting address low	3	03
Number of holding registers high	0	00
Number of holding registers low	1	01
Error checking	-	-

Table 12. Function code 03 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	3	03
Byte count	2	02
Data Holding register 4 HIGH		80
Data Holding register 4 LOW	32768	00
Error check	-	-

The response contains two bytes of data, and the value of the data is 32768 which equals to 65536/2, which in our scenario means that the holding register that contains the data which is used to set an analog output of 0-10V, has been set to 32768 which equals to 5V.

Function code 04, read input register:

- Reads the content of 16-bit input registers
- Specifies the starting input register and the number of input register to read
- In the response one input register corresponds to two bytes of data
- First byte contains the high order bits, and second byte contains the low order bits

As with Function code 01 and function code 02, the function codes 03 and 04 are almost the same. Function code 04 addresses the 16-bit input registers with a reference 3x, so registers 30001 to 39999. Input registers are read only so

they are usually used for analog sensor inputs, like the temperature of a room or the humidity of air.

In the example query shown in Table 13 a query is sent to read input registers 2 and 3, addressed as 1 and 2, which contain data from a sensor which measures the weight in grams of water inside two different containers.

Table 13. Function code 04 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	4	04
Starting address high	0	00
Starting address low	1	01
Number of input registers high	0	00
Number of input registers low	2	02
Error checking	-	-

Table 14. Function code 04 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	4	04
Byte count	4	04
Data Input register 2 HIGH		27
Data Input register 2 LOW	10000	10
Data Input register 3 HIGH		4E
Data Input register 3 Low	20000	20
Error check	-	-

The response shown in Table 14 consists of four bytes of data. The first two bytes contain the data inside input register 2 and the next two bytes contain the data inside input register 3. The data inside the input register 2 is 27 10 and input register 3 is 4E 20, which are 10000 and 20000 in decimal. The input registers contain the mass of water inside a container in grams, so there are 10 kg and 20 kg of water in the two different containers.

Since input registers contain binary data, the values inside them are usually scaled in different ways for better resolution. A device using Modbus Serial usually has a manual where you can see which register or coil has what data stored, and how it is scaled.

Function code 05, write single coil:

- Writes a single coil (1-bit register) as 1 or 0 in other words ON or OFF
- Query sends a byte where the four higher order bits represent the state to be written
- FF 00 sets the coils ON and 00 00 sets the coil OFF
- Can be broadcasted with address field 0, which will write the coil addressed in all the devices in the network
- All other data values are invalid
- Response is an echo of the query

Function code 05 is used to change the state of a coil. It is used to turn on lights, turn off heaters or turn on fans for example. It can only set a single coil with one query. In the example query shown in Table 15 there is a switch that controls a fan on coil 154. It is currently off, and it needs to be turned on. Since the response is an echo of the query, there will not be a separate table for the response.

Table 15. Function code 05 Modbus query and response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	5	05
Coil address high		00
Coil address low	154	9A
Data high	255	FF
Data low	0	00
Error check	-	-

Function code 06, write single register:

- Writes a value inside a single 16-bit holding register
- Query addresses a single register and sends the data to be set to the register
- Can be broadcasted with address field 0, which will write the value sent in the query to all registers with the same address in the network
- Response is an echo of the query

While function code 05 is used to turn ON and OFF switches, function code 06 is used to write a desired data inside a register. Function code 06 reference code is 4x, so it addresses the holding registers. Holding registers are used to set an analog output to a specific value, limits, or targets for the device or even configurations like baud rate which need more data than one bit.

In the function code 03 example the holding register 4 is read and it had a value of 32768 in it, which equals to 5V analog output. Holding register 4 controlled a thermostat in the earlier scenario. Now in this example the analog output is set to 7.5V which would be 49152 in decimal or C000 in hexadecimal. After the query, the slave sets the value of holding register 4 to 49152, which translates to 7.5V in the example analog output. The response is an echo of the query and they are shown in Table 16.

Table 16. Function code 06 Modbus query and response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	6	06
Holding register address high		00
Holding register address low	3	03
Data high		C0
Data low	49152	00
Error check	-	-

Function code 15, Write multiple coils:

- Writes a sequence of coils to ON or OFF state
- Query addresses the starting coil, the number of coils to write in ascending order, the data to write and the byte count
- The coils which are set ON or OFF is specified in the data field of the query
- Broadcast will write the same block of coils in all the devices in the network
- Logic 1 is ON and logic 0 is OFF
- Unused bits in a data byte should be set to zero

While function code 05 could turn a single switch ON or OFF, Function code 15 can do the same thing for a sequence of coils. In the example there is a

sequence of eight coils, and the state of the first five coils in the sequence need to be changed. The starting point and state of the coils is shown in Table 17.

Table 17. Starting point of Function code 15 example

0	1	1	0	0	0	0	1
Coil 7	Coil 6	Coil 5	Coil 4	Coil 3	Coil 2	Coil 1	Coil 0
OFF	ON	ON	OFF	OFF	OFF	OFF	ON

A query is sent to turn coil 0 OFF and coils 1, 2, 3 and 4 ON and do nothing to coils 5, 6 and 7. The query and the response are shown in Tables 18 and 19.

Table 18. Function code 15 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	15	0F
Coil starting address high		00
Coil starting address low	0	00
Number of coils high		
Number of coils low	5	05
Byte count	1	01
Data	30	1E
Error check	-	-

Table 19. Function code 15 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	15	0F
Coil starting address high		00
Coil starting address low	0	00
Number of coils high		
Number of coils low	5	05
Error check	-	-

Function code 15 response will be sent after the data in the query has been written to the coils. The response will not contain the byte count or the data to be written. The starting point of the example should now have changed and is shown in Table 20.

Table 20. The state of coils in Function code 15 after the query

0	1	1	1	1	1	1	0
Coil 7	Coil 6	Coil 5	Coil 4	Coil 3	Coil 2	Coil 1	Coil 0
OFF	ON	ON	ON	ON	ON	ON	OFF

Coils 5, 6 and 7 have not changed and coils 0, 1, 2, 3 and 4 have changed to the states wanted.

Function code 16, write multiple registers:

- Writes a value to a sequence of holding registers
- Query addresses the starting holding register, the number of holding registers to be written in ascending order, the data to write
- Broadcast will write the same sequence of registers in all the devices in the network
- Response doesn’t contain the data to be written

Function code 16 is the equivalent to function code 06 as is Function code 15 to function code 05. In the previous example function code 06 was used to write a value to holding register 4, in this example another thermostat is added to the system that needs to be controlled and it is controlled by the value in holding register 5. We want holding registers 4 and 5 to have the exact same value of 49152, which translated to 7.5V in the analog output of the earlier example. The example query of Function code 16 is shown in Table 21.

Table 21. Function code 16 Modbus query

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	16	10
Holding register starting address high		00
Holding register starting address low	3	03
Number of holding registers high		00
Number of holding registers low	2	02
First register data high		C0
First register data low	49152	00
Second register data high		C0
Second register data low	49152	00
Error check	-	-

This query will write the same value of 49152, C000 in hexadecimal to both, the starting holding register and to the next holding register from the start, so holding registers 4 and 5.

Table 22. Function code 16 Modbus response

Name	Decimal	Modbus RTU (hex)
Slave address	30	1E
Function code	16	10
Holding register starting address high		00
Holding register starting address low	3	03
Number of holding registers high		00
Number of holding registers low	2	02
Error check	-	-

Function code 16 response shown in Table 22 will not send the written data back to the Modbus master. Otherwise, it will echo the query. Broadcasted query will not be responded to. [8.]

2.2.4 Modbus Error Checking Field

Error checking in Modbus can be done by parity checking, or by performing an LRC or CRC calculation on the Modbus messages content and adding the result to the last field of standard Modbus message, the error checking field.

The fields contents are different with Modbus ASCII and Modbus RTU. This chapter will handle the different forms of error checking. First the parity checking then Modbus ASCII error checking and then Modbus RTU error checking.

The standard character in a Modbus message contains 10-bits in ASCII configuration and 11-bits in RTU configuration. The character, or byte consists of a start bit, seven data bits in ASCII and eight data bits in RTU configuration one parity bit and a stop bit. If there is no parity checking implemented, there will be two stop bits. Figures 8 and 9 show the standard character frames of Modbus ASCII and Modbus RTU configuration. Bits are sent from left to right, LSB being the start bit.

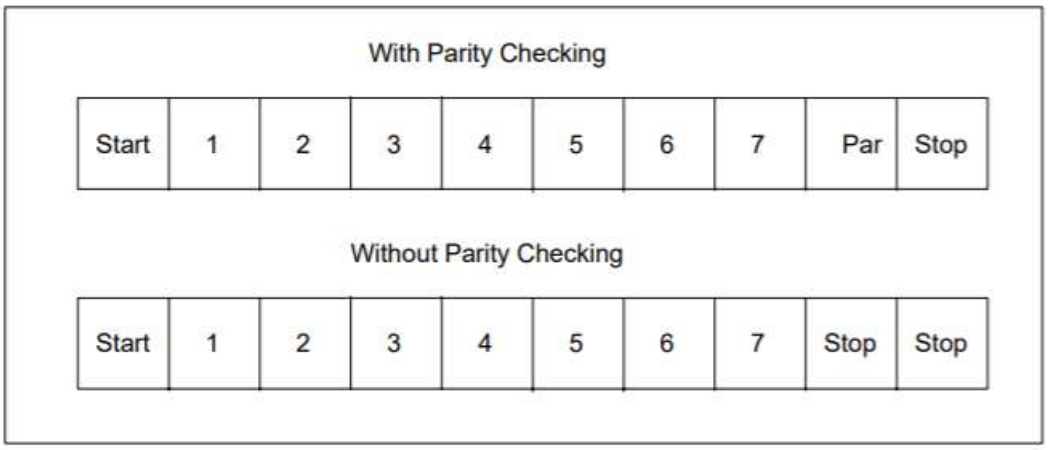


Figure 8. Modbus ASCII character frame [8]

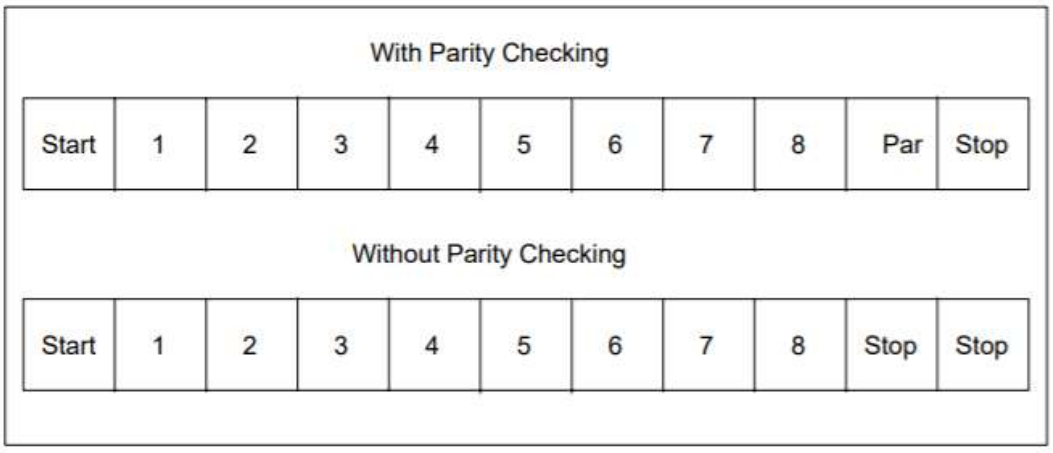


Figure 9. Modbus RTU character frame [8]

Parity checking in Modbus is applied to each character sent. Modbus master generates parity checking in the character and includes it in the message contents before transmission.

In Even or Odd parity checking configuration, Modbus master calculates the amount of even or odd 1-bits in the data field and sets the parity bit accordingly in every frame of every character.

Let's use Modbus RTU character frame as an example, so eight data bits. The sequence of bits is as follows: 10010010. Modbus master then calculates the number of 1-bits and sets the parity bit accordingly. The sequence has three bits set as one, so an odd amount of 1-bits. In Even Parity configuration the parity bit would be set as one in this occasion and in Odd Parity, the parity bit would be set as zero. Then the Modbus slave would calculate the amount of 1-bits in the message received and if the parity bit does not match, the slave will set an error.

To summarize parity checking:

- If Even Parity is used, Modbus master calculates the amount of 1-bits in a character and sets the parity bit as 1 if the amount is odd, and 0 if it is even.

- If Odd Parity is used, Modbus master calculates the amount of 1-bits in a character and sets the parity bit as 0 if the amount is odd, and 1 if it is even.

If No Parity checking is used, the parity bit is transformed into an additional stop bit and no parity check can be made.

Parity checking is used more commonly in Modbus ASCII configuration. Since Modbus RTU, which uses CRC calculation as error checking, is starting to be a lot more used than Modbus ASCII, it unfortunately makes parity checking almost obsolete. Still, Even, Odd and no parity checking options should be

implemented into Modbus controllers because some users might want to or need to have the option to use parity checking.

LRC method is used as error checking in Modbus ASCII configuration. It is implemented in the error checking field, and it contains two ASCII characters, which total to two bytes, even though LRC value is only one byte. Its contents are determined by performing an LRC calculation by the transmitting device on the contents of the message, excluding the ':' colon character and the message termination CR LF characters. The receiving device will then perform the calculation again and if there LRC values do not match, it will set an error.

LRC calculation is done by adding together all the 8-bit bytes in a message and discarding all the carry overs and then performing a 2s complement on the result. As an example, let's think that there is four 8-bit bytes in a message with decimal values of 60 and 40. First, change the decimal values into binary values:

- 60 = 00111100

- 40 = 00101000

Then add the 8-bit bytes together and discard the carry overs:

- 60 = 0011 1100
  40 = 0010 1000

- Which equal to:
  100 = 0110 0100

After adding all the 8-bit bytes together, take 2s complement from it, so invert all the resulting bits and then add 1 to the LSB.

- Inverted 100 = 1001 1011
- Add +1 to the LSB = 1001 1011 + 1 = 1001 1100
- 2s complement 1001 1100 = -100

The LRC value would be -100 or 10011100 in binary and 0x64 in hexadecimal. In this scenario 36 34 would then be written in Modbus ASCII configuration as the LRC value in the error checking field.

In Modbus RTU configuration, error checking is done by CRC calculation. CRC value is a 16-bit binary value added to the error checking field of the Modbus message. As in LRC check, the CRC calculation is also done for the entire Modbus message. The fields contents are determined by performing an CRC calculation by the transmitting device on the contents of the message. The receiving device will then perform the calculation again and if there CRC values do not match, it will set an error.

Calculating the CRC value is a lot more complex, and there will not be an example calculation of it in this thesis. The CRC calculation process is done as follows:

- Set all the bits in one 16-bit register to one
- Take the eight bits of data from a character and exclude the start, parity and stop bits
- XOR the eight bits of data into the least significant byte of the preset 16-bit register
- Shift the 16-bit register towards the LSB with a zero

- Check the value of the LSB, if it is 0, then do nothing. If it is 1 then XOR a preset fixed value to the CRC value

- Repeat the process for eight shifts

- Then take the next characters eight bits of data and repeat until you have gone through all the data bytes in the message

- CRC will be the final value after the process has been done for all of the data bytes in the message

CRC value is then added to the error checking field of a Modbus RTU message, by writing the low order byte first. If our CRC value would be 11110000 00001111, we would write first 0F and then F0 in the error checking field. [9.]

Now we have gone through the basics of Modbus Serial and delved a little deeper into the framing and Modbus messages and its fields. In the next chapter, there will be information on the hardware used for the thesis project.

3 Hardware

The aim for the thesis project was to read data via RS485 from a microbrewery, which uses Modbus RTU as its communication protocol and then send the data to an IoT platform. This chapter will present the Microbrewery and the PLC, Controllino MAXI, used for the project.

As this is an electronics degree thesis, it does not delve too deep into the operations of the microbrewery. The microbrewery chapter will go through the properties of the main board of the AutoLog® PLC used to control the system.

The Controllino chapter will explain why the Controllino MAXI by CONELCOM GmbH was chosen as the PLC for the project and go through some of its properties.

3.1 The Microbrewery

Probably the most fundamental part of the project, the microbrewery, was custom made for Metropolia University of Applied Science by Tankki oy, Finland during winter 2015 and spring 2016. Metropolia uses it to teach brewing technology to its students and for research purposes. Ironically speaking, even though the whole project is about integrating data from the microbrewery into IoT, it is probably the part where the least amount of work was put into. Figures 10 and 11 are pictures of the microbrewery to visualize the main system of the project.

Figure 10. Front view of the microbrewery

Figure 11. Four fermentation tanks of the microbrewery

The system is controlled by AutoLog® 20AN shown in Figure 12. It is a modular PLC from FF-Automation, Finland and the main functions of the microbrewery are controlled by a touch screen control panel. [10.]

The main board AL20AN can be expanded with [11]:

- DI16 digital input board
- RO16 relay output board
- RIO8 digital input / relay out board
- DO32 digital output board
- EXA8/4 analog board (12bit), that can be used to expand number of I/O connections

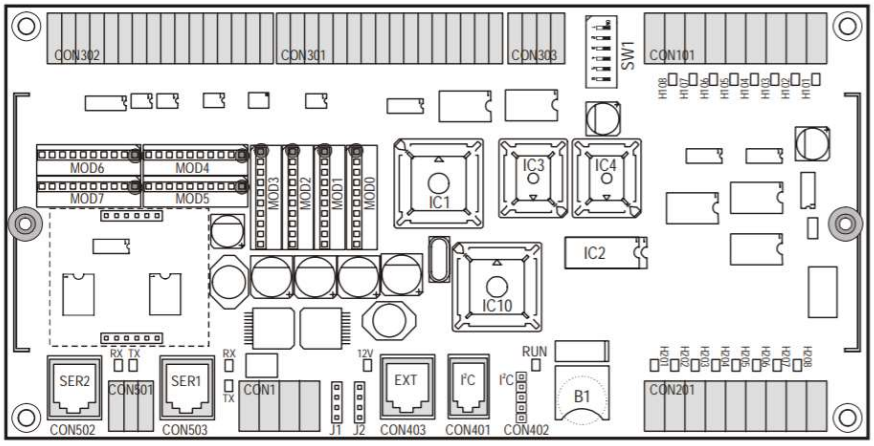


Figure 12. Autolog AL20AN main board

The main boards properties are [11]:

- Input voltage 18 -30VDC / 15.5 - 24VAC or 10 -30VDC / 7.5 - 24VAC from external supply unit
- Eight analog inputs, which can be configured as up to 24 digital inputs with a ratio of one analog input into three digital inputs

- Two 12-bit analog outputs with 0-5V configuration and 0-10V configuration
- Eight 24VDC digital inputs
- Eight 24VDC 1A digital outputs
- I2C interface for the touch screen control panel
- I2C port to connect extension boards
- RS232 serial interface for programming or data communication, SER1
- RS232 serial interface with RS485 conversion module for data communication SER2

3.2 Controllino MAXI

Easy to use, fairly cheap, has RS485 transceiver and can be connected to the IoT. These were the criteria for the microcontroller to use in the project. Controllino MAXI by CONELCOM GmbH shown in Figure 13, meets all of the criteria and more. It is a fully Arduino compatible PLC with open-source software, designed for industrial and DIY projects. And on top of it all, it can be also mounted on a DIN-rail.

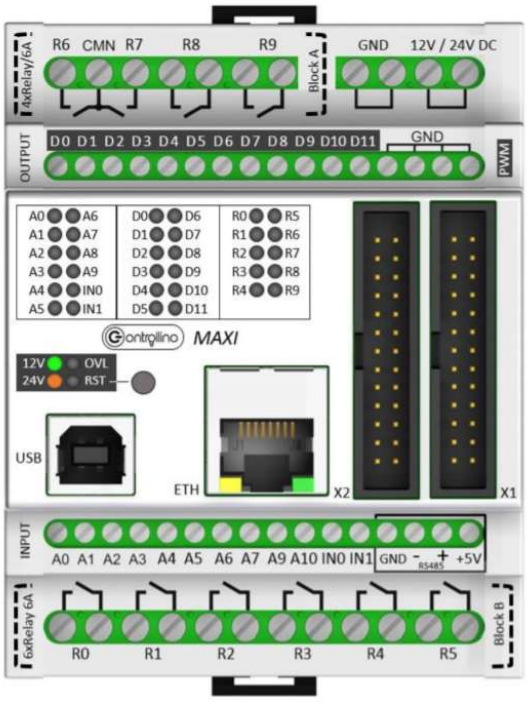


Figure 13. Controllino Maxi [12]

It is basically an Arduino in a PLC form for automation applications. It can be programmed with software's that are compatible with Arduino like Labview, Matlab, Atmel Studio and of course Arduino IDE.

The PLC needs a supply voltage of 12VDC or 24VDC which makes it perfect for automation application. It is equipped with:

- 12 configurable analog or digital inputs

- 6 digital inputs
- 2 analog inputs 0-10V
- 2 interrupt inputs
- 8 digital PWM outputs 2A
- 2 analog outputs 0-10V
- 10 relay outputs
- I2C, SPI, RS485 and ethernet interface
- 2 serial TTL interface
- A built-in real-time clock

It also has pin headers which connect straight to the microcontroller on a 5V logic level. [12.]

The Controllino MAXI might have been a bit too much for the project, since only RS485 and ethernet was needed, but thinking ahead, Controllino MAXI is the perfect solution for the microbrewery if something needs to be implemented later.

4 Software Platforms

This chapter introduces the software platforms used in the thesis project and explains why they were chosen for the project.

4.1 Arduino IDE

The open-source software, Arduino Integrated Development Environment or Arduino IDE makes writing code fast and easy. Since Controllino MAXI is fully Arduino compatible, it was a no-brainer to use Arduino IDE as the programming platform.

There are probably thousands of third-party libraries for Arduino IDE which makes writing working code very fast. Especially things like configuring your device for Modbus- or ethernet protocol might be really time consuming. With Arduino IDE you can just find libraries online and add them to your project through the software with a few clicks.

It also has an easy-to-use serial monitor to debug your code and test what you have implemented, so there is no need for external terminal emulator software like PuTTy or TeraTerm. Uploading the code is also very easy and fast, connect your Arduino compatible device via USB to your computer, compile the code and upload it to your device. [13.]

4.2 Ubidots STEM

It was easy to pick the programming platform to use in the project but picking the IoT platform turned out to be more difficult. There were several options to choose from and even more options if some money would have been used on the platform. To narrow down the options from which to choose, decisions were made that the IoT platform for the project should be free or at least free to some extent.

It was also wanted that the platform would be easy to configure with minimal knowledge of IoT. Options like Blynk and Arduino IoT Cloud emerged but there were some problems with both. Arduino IoT Cloud needed a board, which is compatible with the platform and Controllino MAXI was not one of them so that was a no for Arduino IoT Cloud. Blynk was a good candidate with possibility to implement the project onto a phone app, but I have been working with Blynk before, and the free version of it is very limited with the widgets you can use.

Then Ubidots STEM was found. An IoT platform by Ubidots for educational purposes and non-commercial use, created in 2018. For free, Ubidots STEM users can sign-up up to three devices and 10 variables per device. With Ubidots STEM you can also send 4000 “dots” per day to the platform. A dot is a data point containing a value and timestamp. You can have up to three dashboards and you can place up to 10 widgets per dashboard. For a free platform Ubidots STEM was a very solid platform for the project. With more money invested, you can always get a better one though.

Ubidots STEM is limited as well, but the ease of use and the configurability of the dashboard made it enjoyable to use, so it was decided to use it as the IoT platform. In the project it is not needed to send data too often to the IoT, so the 4000 dots limit won’t be a problem. Every five minutes you can send almost 14 dots to the platform. There is also a limitation of 10 variables per device, so basically every five minutes you can send a dot to each of the variables you use and still stay withing the 4000 dots per day limitation. Brewing beer is not very time critical, so getting data every five minutes will work just fine. Of course, you can configure the code so that some of the more time critical variables get data sent to them more often. [14.]

5 The Project and its Results

The task was to implement an IoT solution for the microbrewery in Metropolia University of Applied Sciences food laboratory. RS485 had to be used as the serial communication interface with Modbus RTU as the communication protocol. The main goal for the project was to log data from the Brix-meter of the brewery into a graph in the cloud. The work started with the physical layer of the project, connecting the Controllino into the microbrewery.

First, the SER2 port, as in the datasheet, or COM2 port, as labelled on the tank itself shown in Figure 14, was configured for the use of RS485 with DIP-switches, SW1 in Figure 12.

Then RS485 signals A, B and GND were connected from the SER2 port into the Controllino MAXI PLC. Next, the supply voltage of 24VDC from the external power supply was connected to power the Controllino MAXI. The PLC was also connected into a router provided for the project through its ethernet port.

Figure 14. SER2 port of the microbrewery

After the electrical connections were made, the Modbus holding register addresses, seen in Appendix 3, were studied and the programming of the Controllino MAXI PLC begun. Figure 15 shows the wired up Controllino MAXI

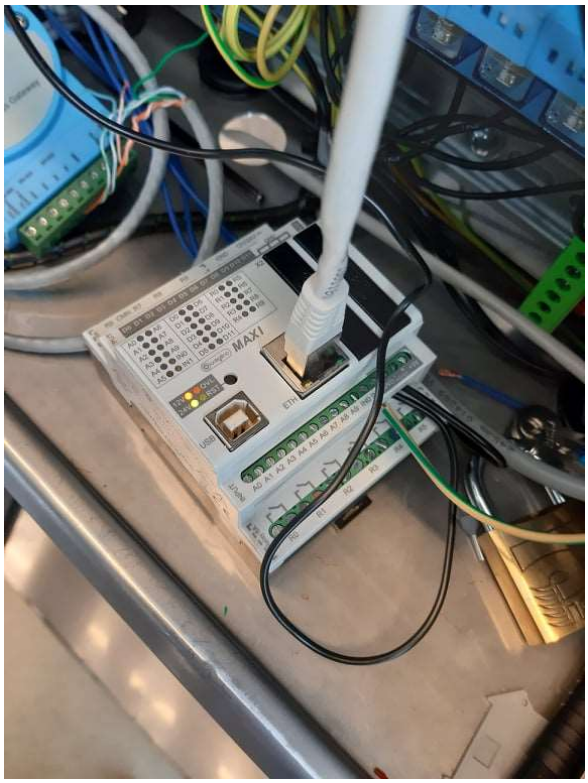


Figure 15. Controllino MAXI wired up and ready to be programmed

The most time-consuming part of the project was programming the hardware. What happens in the code will be explained in this chapter and the code will be added in full as Appendix 3. The code is divided into five parts, or tabs in Arduino IDE to make it easier to debug and read the different parts of the code.

First is the tab for the included libraries, defines and global variables of the code. The included libraries are ready-made libraries to make using Controllino, ModbusRTU and Ubidots quite a bit easier. The possibility to use these libraries is the reason Arduino IDE is so popular among hobbyists in my opinion.

In the first tab, Controllino MAXI is set as the Modbus master and the addresses for the slave and master is defined. The COM port is set and everything

necessary for the ethernet and Ubidots STEM is defined. Then a couple of global variables and the ModbusQuery struct used in the later parts of the code are declared.

The second part of the code is the setup, where the Baud rate is set the ethernet is started. The Modbus queries are also set there. Every holding register of the microbrewery was wanted to be read, so it would be easier to modify the code when someone wants to upload data to the cloud. It was found out that only 29 holding registers could be read per query, so in the setup, five different queries were set with four of them reading 29 holding registers and the last of them reading five. The variables for the loop of the code were also set in this part.

In the loop part the queries are sent to the Modbus slave and the contents of the holding registers are combined into a single array called tankkiRegister, indexed as in the holding register file in Appendix 2. The contents of the holding registers are also printed to the serial monitor for debugging purposes. After sending the queries the loop will call the SendToUbidots function where the received data is sent to Ubidots STEM.

If the microbrewery will ever have this solution in use, the user should consider adding a lot more delay before sending the data of the holding registers into Ubidots STEM because of the limitations of the dots sent to the platform per day. It is not necessary to get new data from the microbrewery to the IoT platform too often because the brewing process is not that time critical.

The fourth part is the part which can be modified to choose and change what holding registers values are sent to the IoT platform. It consists of defining the variable labels seen in the Ubidots STEM and then the function SendToUbidots. In the function there are declarations of variables with values assigned to them from scaling the values in tankkiRegister array and then sending the variables into Ubidots STEM. Comments have been added to the fourth tab, which explain how to modify the SendToUbidots function as the user needs.

The final tab of the code contains only the two different scaling functions used to get usable values out from the holding registers. In Appendix 2, there are three different types of scaling for the values inside the holding registers:

0. No scaling

1. Scaling 1 = if you get a value of 1250 as a temperature you need to deduct 1000 from it and divide the number by 10. Then you get 25 as the temperature, Units are shown in the Appendix 2.

2. Scaling 2 = just divide the value you get by 100, so if you get 700 as the pH reading, divide it by 100 and you get seven.

After programming the Controllino MAXI the configuration of Ubidots STEM platform was next. The plan was to add as much data and graphs of different sorts on the dashboard as possible, because one goal of the thesis was to write a user manual on how to configure the platform and the values sent to it. Having a lot of different data and configures on the Ubidots STEM side would make it easier to go through the different options on how to set up the platform the way the user wants.

Figure 16. Ubidots STEM dashboard for the project

When the microbrewery is powered on, the Controllino MAXI will start querying the Modbus holding registers, stores the data in a buffer variable and sends it into the Ubidots STEM platform. In the project the temperature of the four fermentation tanks as a graph and a numerical value, the pH value and the temperature of the kettle, the temperature of the flushing water and of course the Brix value, which was the whole purpose of the project is sent to the dashboard of Ubidots STEM. The dashboard can be seen in Figure 16.

6 Conclusions

The first goal of the thesis project was to implement the possibility to read and log Brix value to an IoT platform from the microbrewery of Metropolia University of Applied Sciences. Unfortunately, it is not allowed to use the microbrewery so tests could not be made to provide any other value than zero as Brix value from the Brix meter. The second goal was to create a manual for anyone who wishes to set and configure the solution for their own use.

Even though both goals of the thesis project were successfully finished, there is a lot of room for improvement. There are arbitrary ways to improve it like cleaning the code and installing the hardware more properly, but its functionality can also be improved a lot.

Only the capability to read from the holding registers of the microbrewery was implemented in the thesis project but writing to the registers and controlling the microbrewery from the cloud is also possible to add. Adding money or building your own IoT platform would be a good way to improve the thesis project as well. As for second goal, the manual was made as simple to read as possible and configuring the code to the user's own preference is made easy as well.

Making things remotely and easily accessible is the future of the modern world.

References

1 Lammert Bies. Practical information about implementing RS485. https://www.lammertbies.nl/comm/info/RS485 Accessed 11 August 2021.

2 B+B Smartwork, BASICS OF THE RS485 STANDARD. https://www.bb-elec.com/Learning-Center/All-White-Papers/Serial/Basics-of-the-RS485-Standard.aspx Accessed 19 August 2021

3 Jason Kelly, CUI DEVICES, RS485 Serial Interface Explained. https://www.cuidevices.com/blog/RS485-serial-interface-explained Accessed 11 August 2021

4 OPTCORE 2018 (updated 2021), What is the difference between RS-232, RS-422, and RS-485? https://www.optcore.net/difference-between-rs-232-rs-422-and-rs-485/ Accessed 23 August 2021

5 William (Bill) L. Mostia, Jr., P.E., principal engineer, WLM Engineering Co. 2019, Introduction to Modbus. https://www.controlglobal.com/articles/2019/introduction-to-modbus/ Accessed 23 August 2021

6 Technical support note MTL Fieldbus network solutions, 2019, Introduction to Modbus. https://www.mtl-inst.com/images/uploads/TSN_MTL838C_Modbus_Rev_1.pdf Accessed 23 August 2021

7 Modbus Tools, Protocol Description. https://www.modbustools.com/modbus.html Accessed 6 September 2021

8 Bruce Cyburt, Automation.com, 2012, Introduction to Modbus. https://www.automation.com/en-us/articles/2012-1/introduction-to-modbus Accessed 2 September 2021

9 MODICON, Inc., Industrial Automation Systems 1996, Modicon - Modbus Protocol Reference Guide. https://modbus.org/docs/PI_MBUS_300.pdf Accessed 5 September 2021

10 Santeri Tenhovirta, 2016, Studies on Operation and Carbohydrate Yield in a Microbrewery, Engineer's thesis. https://www.theseus.fi/bitstream/handle/10024/116123/tenhovirta_santeri.pdf?sequence=1&isAllowed=y Accessed 6 September 2021

11 FF-AUTOMATION OY, 2006, AL 20AN Programmable Logic Controller Instruction Manual.
http://www.ff-automation.com/download/Documents/English/AutoLog_Manuals/ManAL20AN29122006.pdf
Accessed 6 September 2021

12 CONELCOM GmbH, 2018, Controllino Instruction Manual "CONTROLLINO" MINI, MAXI and MEGA.
https://www.controllino.com/wp-content/uploads/2019/02/CONTROLLINO-Instruction-Manual-V1.5-2018-12-14.pdf
Accessed 6 September 2021

13 Arduino, Introduction.
https://www.arduino.cc/en/Guide/Introduction
Accessed 14 September 2021

14 Ubidots, Ubidots STEM.
https://ubidots.com/stem/
Accessed 14 September 2021

Appendices

Appendix 1. User Manual

Contents

1. Set up Ubidots STEM account
2. Set up Controllino MAXI as the Ubidots STEM device
3. Modify the Arduino Code
    - Modify the variables sent to Ubidots STEM
    - Flash Arduino code into Controllino
4. Configure Ubidots STEM platform

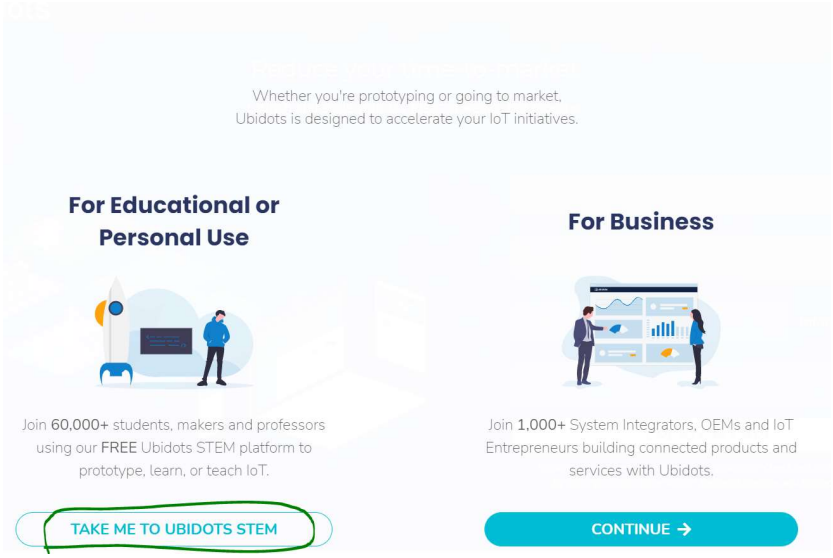
Tools needed:

- USB 2.0 A – B cable
- Arduino IDE
- Internet access

1. Set up Ubidots STEM account

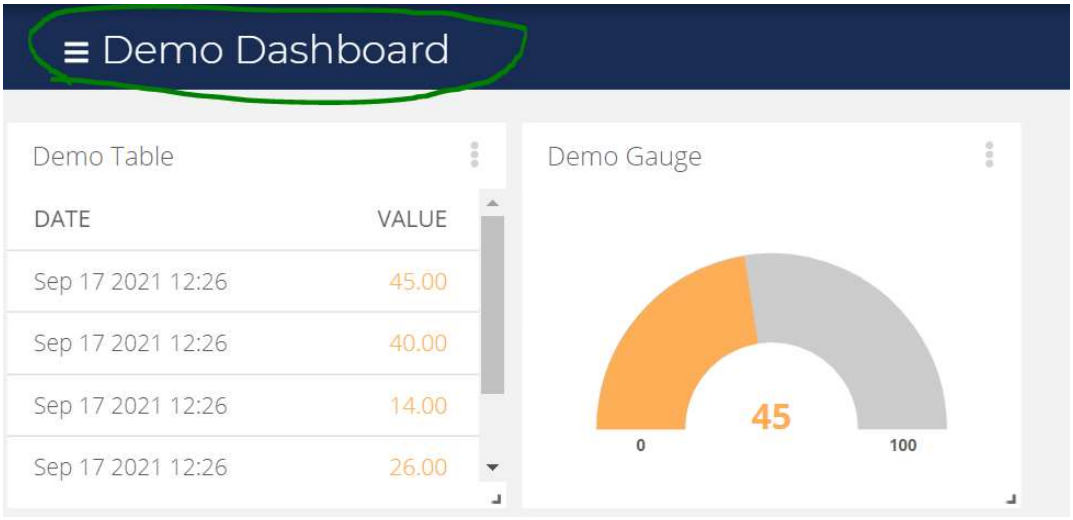
1. Go to https://industrial.ubidots.com/accounts/signup_industrial/

2. A popup comes up, click on the “Take me to Ubidots STEM”

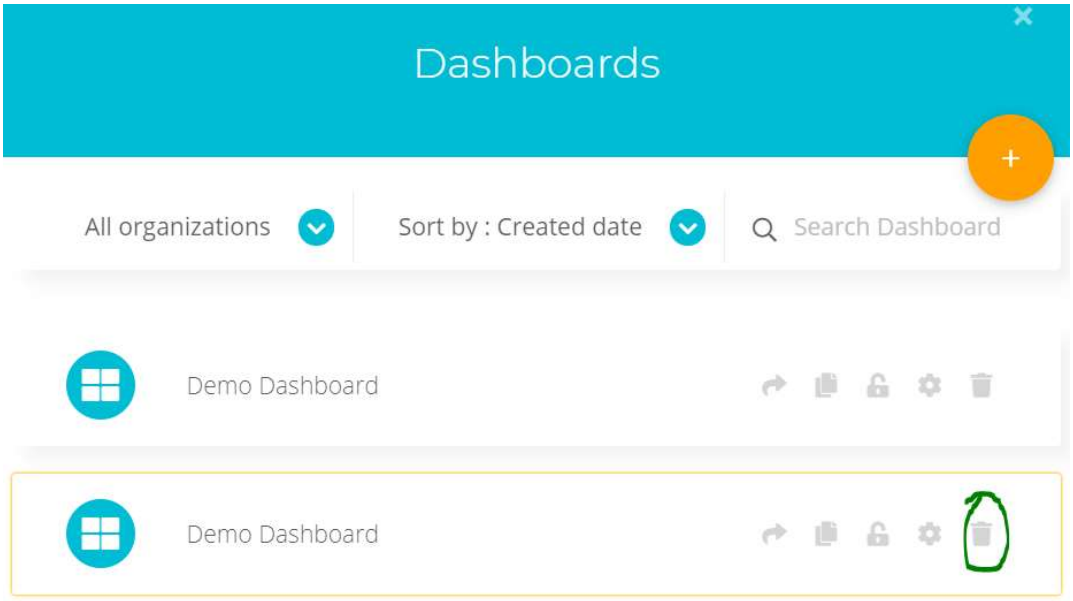


3. Make an account, check “My IoT project is for personal, non-commercial use” box and click “Sign up for free”.

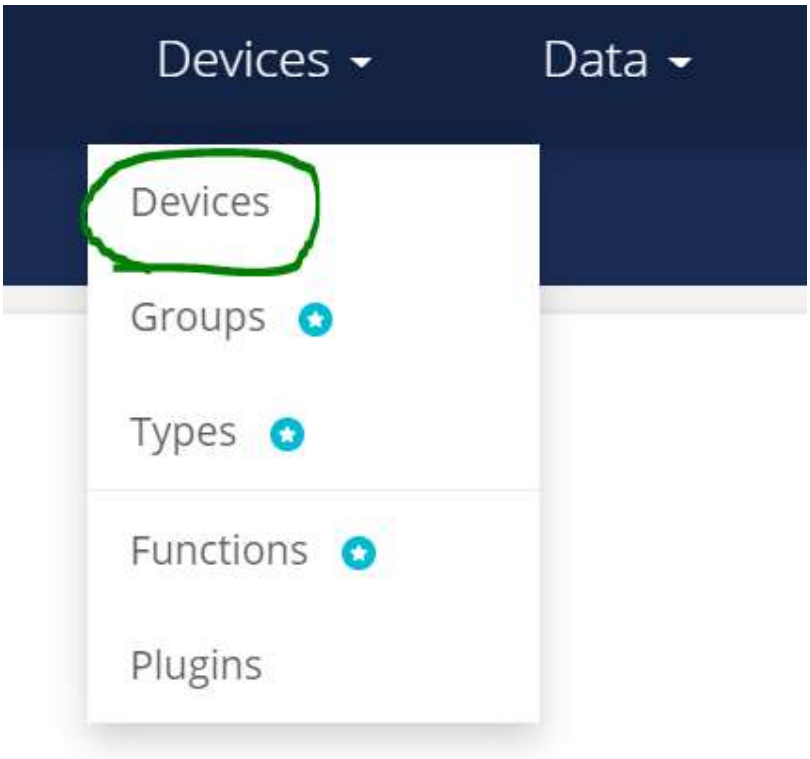
4. Go through the quick tutorial and then delete the demo dashboards by clicking next to the “Demo dashboard” text



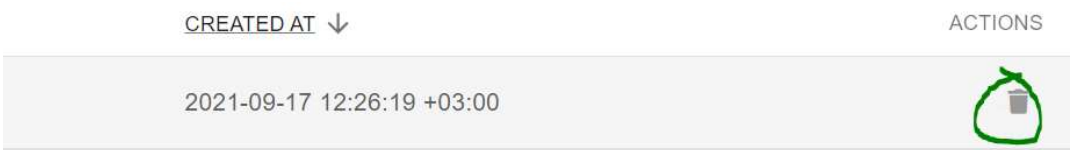
5. Click on the trashcan icon and confirm with delete on the popup which shows up to delete the demo dashboards. Do it for all the demo dashboards.



6. Go back and click on the devices popup menu. Click devices.

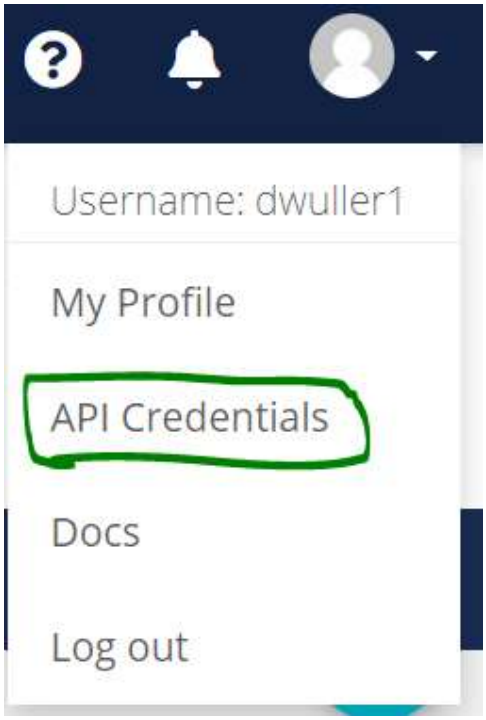


7. Delete the demo device by clicking on the trash can and confirming with delete.

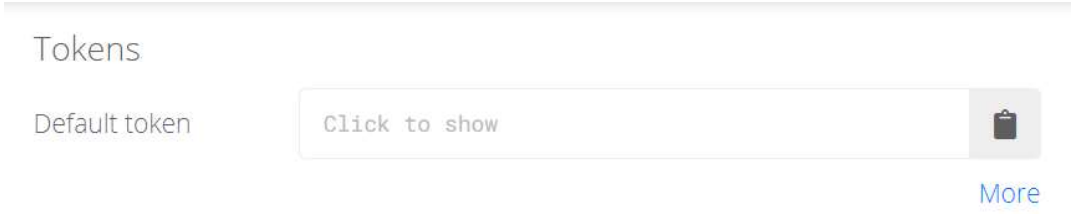


8. Now you have a fresh Ubidots STEM account with everything extra deleted.

9. Next click on the user icon on the main toolbar and click API credentials from the popup menu



10. API credential menu will show up, here you can check and copy the Default token for your Ubidots STEM account, which will be needed in setting up Controllino MAXI



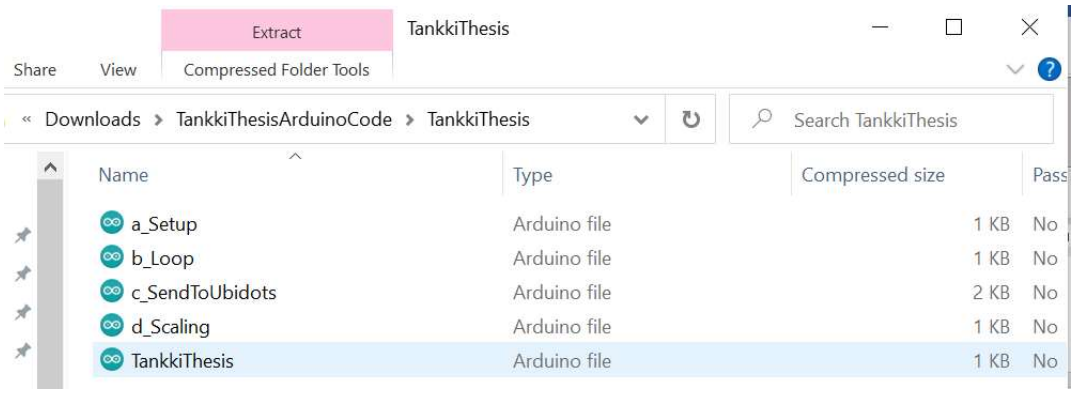
2. Set up Controllino MAXI as the Ubidots STEM device

1. If you don't already have Arduino IDE software, download it from this link: https://www.arduino.cc/en/software

2. Download the Arduino code named written for this project located in: **oma.metropolia.fi Olutprosessi innovaatioprojekti 2020 workspace** in the folder Arduino Code. The file is named TankkiThesisArduinoCode.zip

3. Extract the folder into a location you want

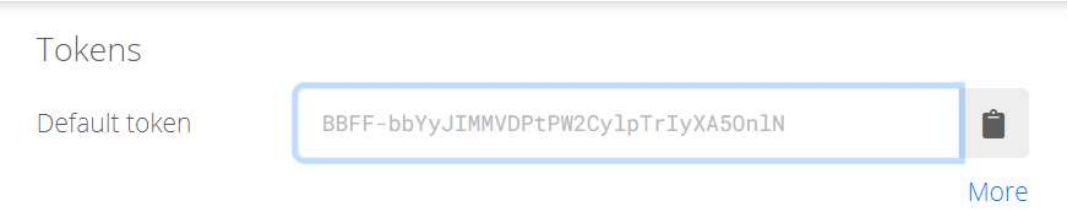
4. The contents of the folder should look like this:

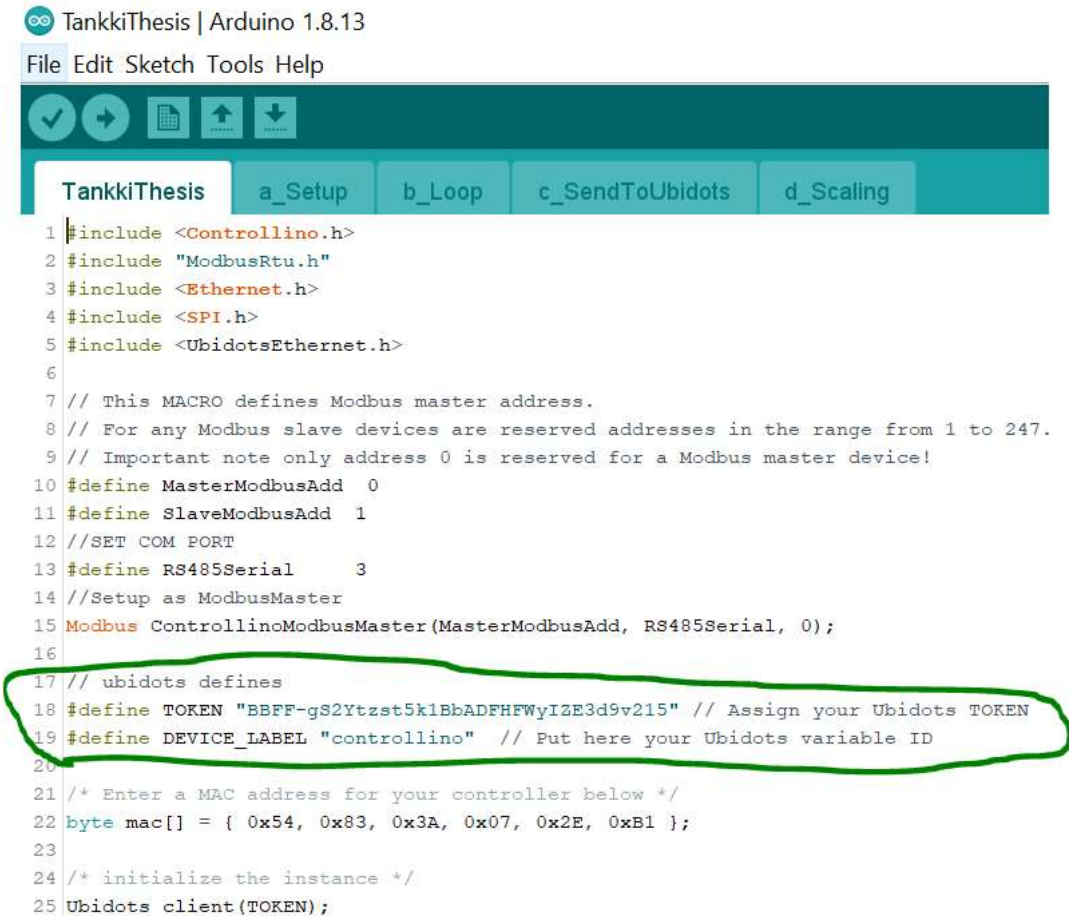


5. Open the TankkiThesis Arduino file

6. On the first tab called TankkiThesis, on the lines 18 and 19 are the definitions for your Ubidots Default token and Device label.

7. Change the Default token to the token you can copy from your freshly created Ubidots STEM accounts API credentials **NOTE. The default token is different for every account**

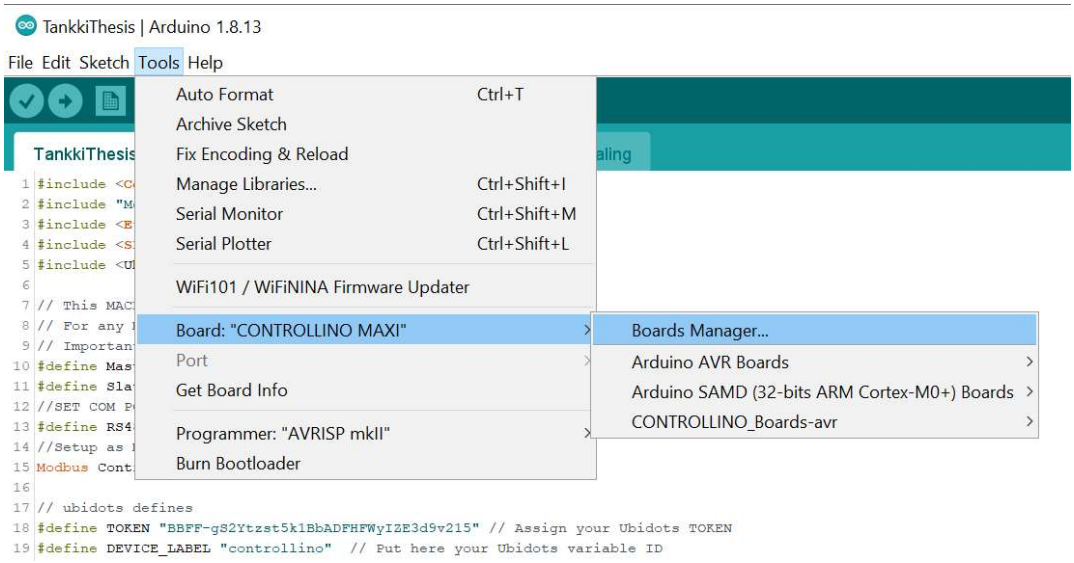





```
#include <Controllino.h>
#include "ModbusRtu.h"
#include <Ethernet.h>
#include <SPI.h>
#include <UbidotsEthernet.h>

// This MACRO defines Modbus master address.
// For any Modbus slave devices are reserved addresses in the range from 1 to 247.
// Important note only address 0 is reserved for a Modbus master device!
#define MasterModbusAdd  0
#define SlaveModbusAdd  1
//SET COM PORT
#define RS485Serial     3
//Setup as ModbusMaster
Modbus ControllinoModbusMaster(MasterModbusAdd, RS485Serial, 0);

// ubidots defines
#define TOKEN "BBFF-gS2Ytzst5k1BbADFHFWyIZE3d9v215" // Assign your Ubidots TOKEN
#define DEVICE_LABEL "controllino"  // Put here your Ubidots variable ID

/* Enter a MAC address for your controller below */
byte mac[] = { 0x54, 0x83, 0x3A, 0x07, 0x2E, 0xB1 };

/* initialize the instance */
Ubidots client(TOKEN);
```

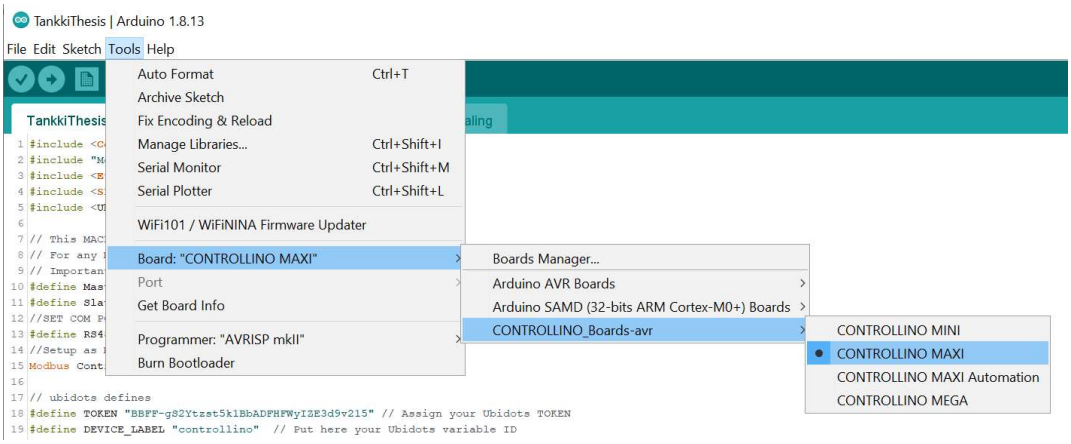
8. To be able to work with the Controllino devices in the Arduino IDE, you will need to install the Controllino board using the preconfigured Arduino Board Manager.

9. Go to Tools -> Board -> Boards Manager



10. A boards manager popup will show up. Search for Controllino and install it

11. When the board is installed, select it by going Tools -> Board -> Controllino_Boards-avr -> Controllino MAXI



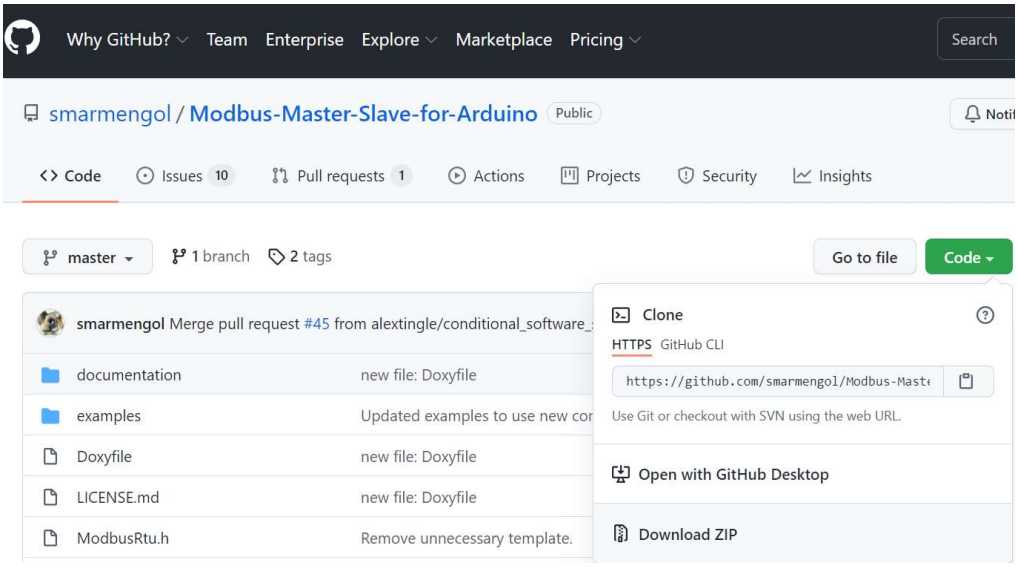
12. Then install these libraries. Install the ModbusRTU library as well, as it is needed for Controllino MAXI to be able to communicate with the microbrewery

    - Ubidots Ethernet library
      https://github.com/ubidots/ubidots-arduino-ethernet

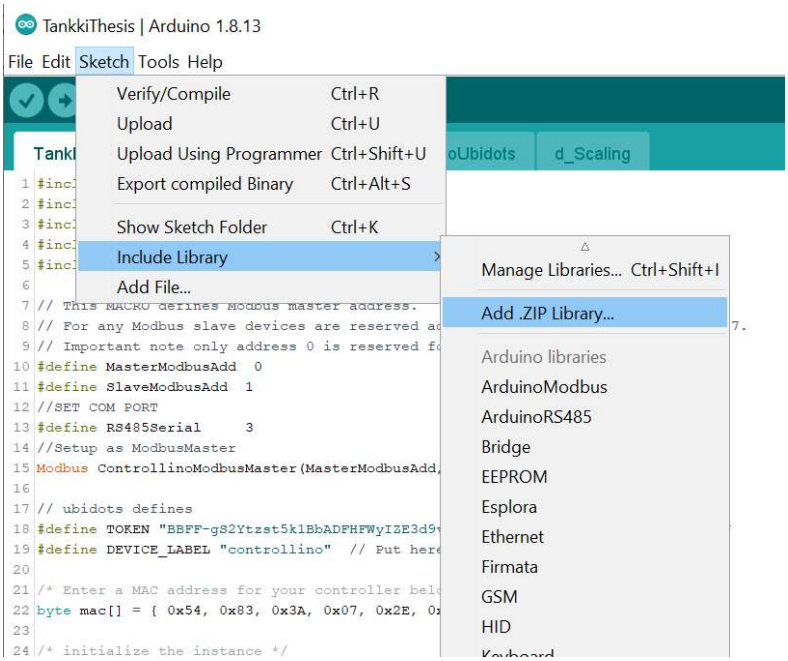
    - Controllino library
      https://github.com/CONTROLLINO-PLC/CONTROLLINO_Library

    - ModbusRTU library
      https://github.com/smarmengol/Modbus-Master-Slave-for-Arduino

13. To install the libraries, go to the link -> take your cursor to the green code popup menu -> click download ZIP. DO NOT EXTRACT THE ZIP-FILE



14. To add the downloaded libraries into Arduino IDE, go to Sketch -> Include Library -> Add .ZIP library. Locate and add the downloaded .ZIP files.



15. Finally close and restart the Arduino IDE.

16. Now we have everything setup for the Controllino MAXI to be able to send data to Ubidots STEM.

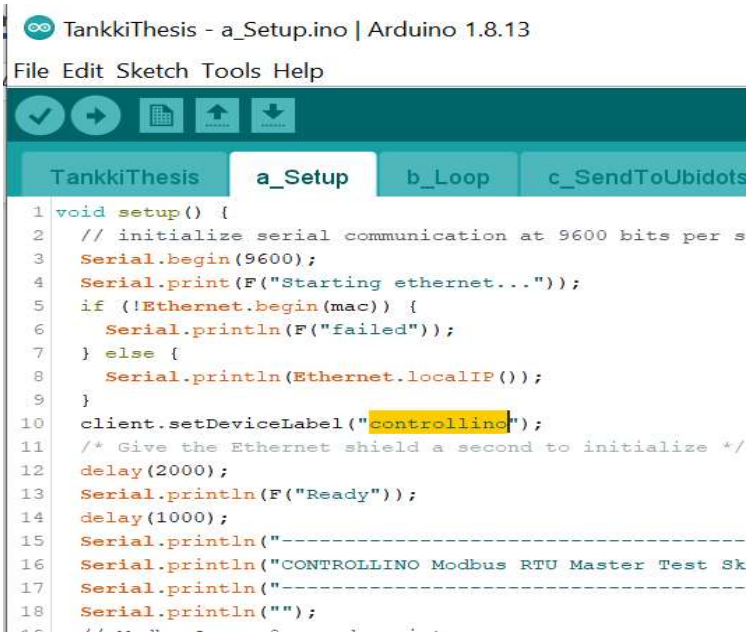
3. Modify Arduino Code

1. To modify the device name and the variables sent to your Ubidots STEM platform, the code needs to be adjusted.

2. We don't need to do anything yet to the Ubidots STEM platform. When the Controllino MAXI is configured and powered on, It will send and create the Device and the Variables automatically into Ubidots STEM.

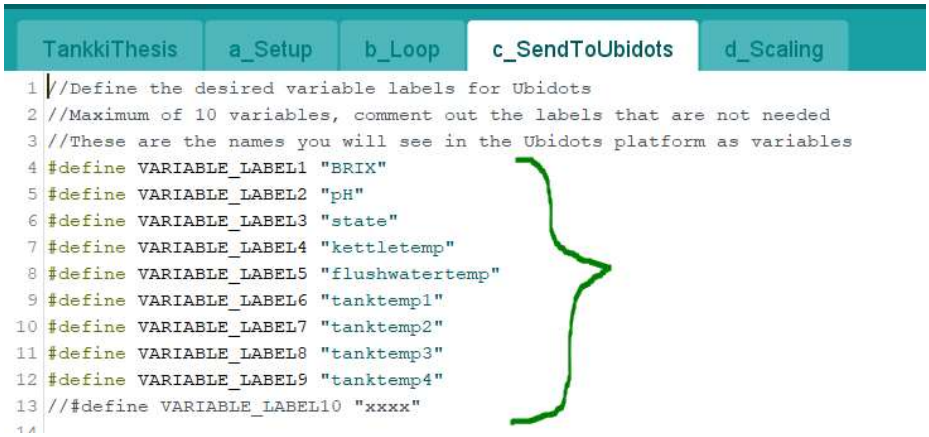
3. To set the name for the device you want to have in Ubidots STEM platforms side. Open **TankkiThesis** Arduino file.

4. Click on the tab **a_Setup** and look for line 10. Modify the name inside the quotation marks as you wish. This is the name of the device you will see on Ubidots STEM platform side.



5. To modify the variables sent to Ubidots STEM platform click on the tab **c_SendToUbidots**

6. From line 4 to 13 you can modify the name of the variable seen in Ubidots STEM side. Just change the name inside the quotation marks as you wish




```
1  //Define the desired variable labels for Ubidots
2  //Maximum of 10 variables, comment out the labels that are not needed
3  //These are the names you will see in the Ubidots platform as variables
4  #define VARIABLE_LABEL1 "BRIX"
5  #define VARIABLE_LABEL2 "pH"
6  #define VARIABLE_LABEL3 "state"
7  #define VARIABLE_LABEL4 "kettletemp"
8  #define VARIABLE_LABEL5 "flushwatertemp"
9  #define VARIABLE_LABEL6 "tanktemp1"
10 #define VARIABLE_LABEL7 "tanktemp2"
11 #define VARIABLE_LABEL8 "tanktemp3"
12 #define VARIABLE_LABEL9 "tanktemp4"
13 //#define VARIABLE_LABEL10 "xxxx"
```

7. Then declare variables inside SendToUbidots() function and store a value of a holding register into it. You can see what each holding register contains in Appendix 2.

8. All of the data inside the holding registers are stored in the array called tankkiRegister with tankkiRegister[0] being Holding register A1, and tankkiRegister[120] being Holding register A121.

```
/* examples =
 *  You want to add the variable: temperature of pH measure to Ubidots
 *  define a Variable label not in use with a name suited for it
 *  #define VARIABLE_LABEL10 "pHtempMeasure"
 */
void SendToUbidots() {
  float BRIX = scaling2(tankkiRegister[12]);              //Array tankkiRegister[] contains data from all of TANKKI's holding registers
  float pH = scaling2(tankkiRegister[13]);                //Check from Tankkis register map to see which register contains which value
  float kettletemp = scaling1(tankkiRegister[0]);         //tankkiRegister[0] = Holding register 42150
  float flushwatertemp = scaling1(tankkiRegister[2]);     //tankkiRegister[x] = Holding register x + 42150
  int state = tankkiRegister[86];                         //Define your variables and check the scaling if needed in Tankkis register map
  float tanktemp[4];                                      //functions are scaling1(tankkiRegister[x]); and scaling2(tankkiRegister[x]);
  tanktemp[0] = scaling1(tankkiRegister[22]);             //Use float or double values if decimals are needed
  tanktemp[1] = scaling1(tankkiRegister[23]);
  tanktemp[2] = scaling1(tankkiRegister[24]);
  tanktemp[3] = scaling1(tankkiRegister[25]);
```

9. Use floating point values if you need decimals for the value.

10. Check which type of scaling you need from Appendix 2.

    a. If the The holding register has Skaala as 1 use scaling1() and pass the corresponding tankkiRegister value into it

       Example: To send the kettles temperature into Ubidots.

       In Appendix 2. You can see that the kettle temperature is stored in Holding register A1 and uses scaling 1.

       Declare a floating point variable -> pass tankkiRegister[0] into scaling1 function -> store the returned data into the floating point variable.

KÄYTTÖOHJE SIVU 1/3

Osoite	Skaala	Sisältö	[yksikkö]
A 1	1	Kattilan lämpötila TE1 [°C]	

```
float kettletemp = scaling1(tankkiRegister[0]);
```

    b. If the The holding register has Skaala as 2 use scaling2() and pass the corresponding tankkiRegister value into it

       Example: To send the pH value into Ubidots.

       In Appendix 2. You can see that the pH measurement is stored in Holding register A14 and uses scaling 2.

Declare a floating point variable -> pass tankkiRegister[13] into scaling2 function -> store the returned data into the floating point variable.

A 14	2	pH-mittaus [0.00 - 14.00]

```
float pH = scaling2(tankkiRegister[13]);
```

c. If the The holding register has Skaala as 0 just store the value from tankkiRegister array linked to the corresponding Holding register into the variable

11. Finally add the variables where you have stored the data from tankkiRegister array into the add function and send the data to Ubidots.

```
client.add(VARIABLE_LABEL1, BRIX);
client.add(VARIABLE_LABEL2, pH);
client.add(VARIABLE_LABEL3, state);
client.add(VARIABLE_LABEL4, kettletemp);
client.add(VARIABLE_LABEL5, flushwatertemp);
client.sendAll();

client.add(VARIABLE_LABEL6, tanktemp[0]);
client.add(VARIABLE_LABEL7, tanktemp[1]);
client.add(VARIABLE_LABEL8, tanktemp[2]);
client.add(VARIABLE_LABEL9, tanktemp[3]);
//client.add(VARIABLE_LABEL10, yourVariable);
client.sendAll();
```

12. The data from the holding register A14 containing the pH measurement is stored in the variable pH. The VARIABLE_LABEL2 is defined as “pH”

13. In Ubidots STEM you will now have a variable called pH and it contains the data from the Arduino variable called pH which stores the data from Holding Register A14.

14. Variable label will be the name of the variable seen in Ubidots STEM side. Double check that the variable label you want will contain the right data.

15. **NOTE1 THE MAXIMUM NUMBER OF VARIABLES THAT CAN BE SENT TO UBIDOTS STEM IS 10 WITHOUT PAYING EXTRA.**

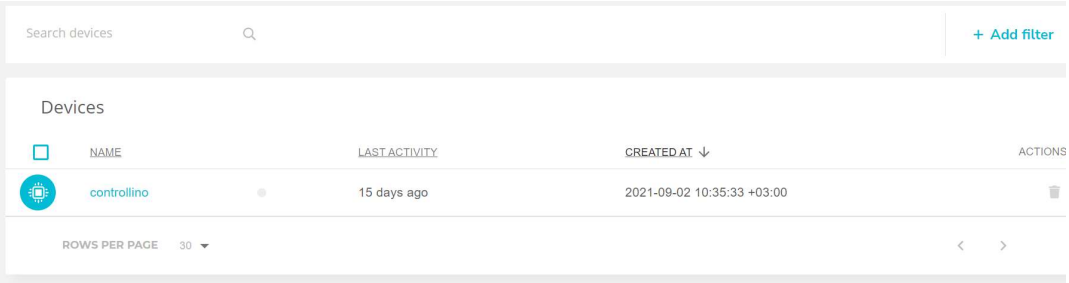
16. **NOTE2 ONLY 5 VARIABLES CAN BE SENT TO UBIDOTS WITH ONE client.sendAll() FUNCTION**

4. Configure Ubidots STEM platform

1. Sign in to your Ubidots STEM account.

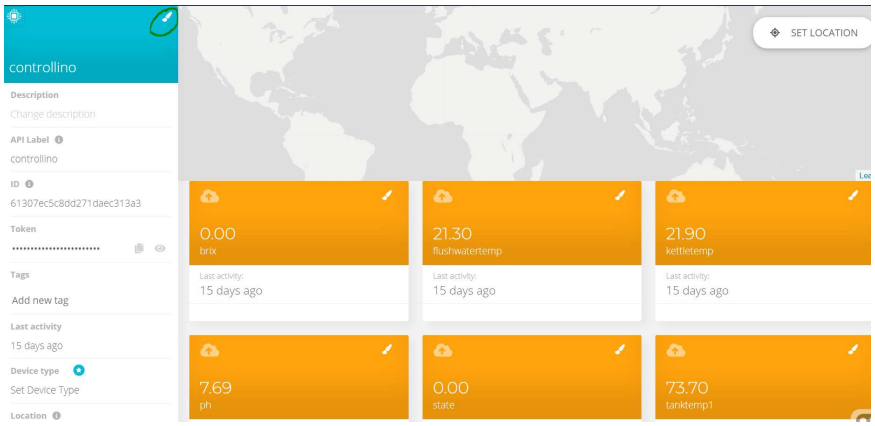
2. Go to your devices from the Devices popup menu.

3. If the Controllino MAXI has been up and running, it should have added a new device to your device list named ”controllino”



4. Click on the name of the device to open its menu

5. You should see all the variables you have added to the Arduino Code and sent to Ubidots STEM



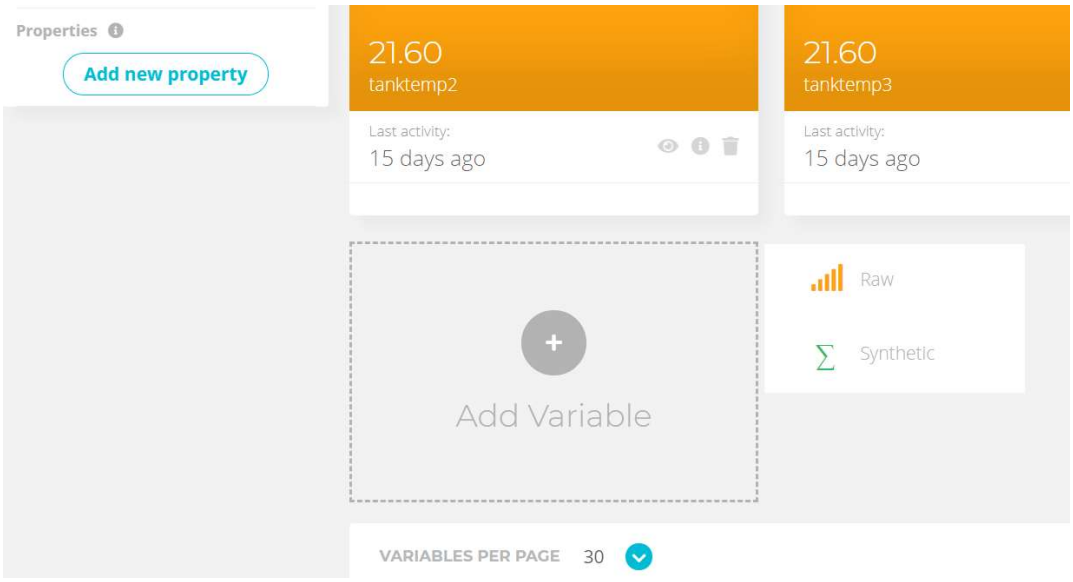
6. Here you can change the name of your device by clicking on the brush

7. You can also change the name of the variables by clicking on the corresponding brush

8. There is also an add variable button when you scroll down the menu, there you can add own raw variables or synthetic variables.

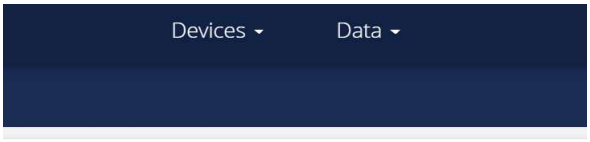
    a. Raw variables are not needed to add, because by sending a new variable configured in the Arduino code into Ubidots, it automatically creates a new variable into the list

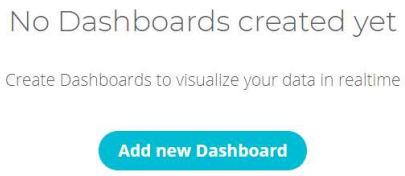
    b. You can take the data from a raw variable and create a synthetic variable from it. This way you can modify the value with equations. This is not needed for this application because the scaling is done in the Arduino code. **Note. Adding synthetic variables will count to the 10 variable limit.**



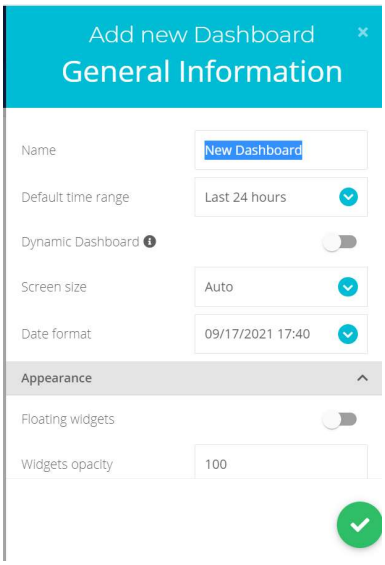
9. To use the data stored in the variables in Ubidots STEM you will need to make a new dashboard

10. Go to Data -> Dashboard and click add new dashboard



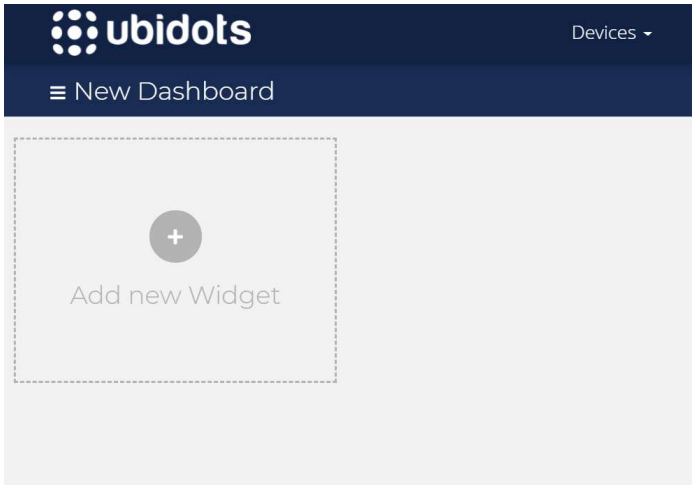


11. A popup window will show up where you can edit the dashboards settings

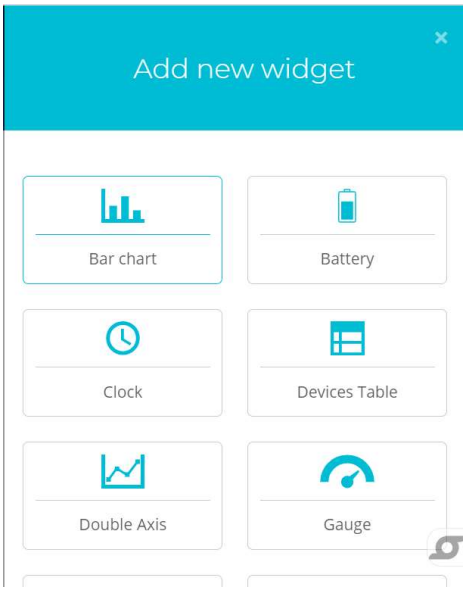


12. After you have configured the settings for the dashboard as you like, click on the confirm mark to create the dashboard.

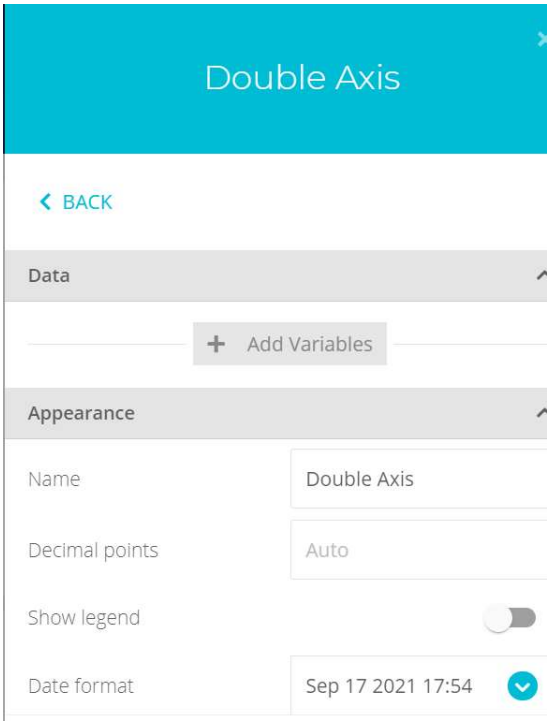
13. You will be taken to your newly created dashboard where you can add widgets to the dashboard. There are a lot of different widgets you can use ranging from graphs to charts and just plain numeric values



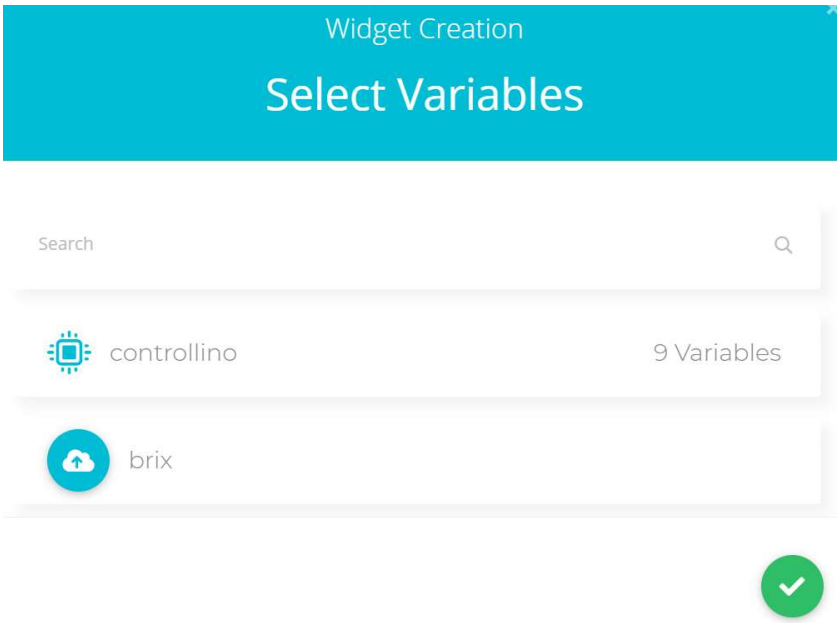
14. To add a line graph, click on the “Add new widget” button and from the popup menu select double axis.



15. A new menu will show up where you can configure your graph the way you want.



16. To link a variable sent to Ubidots STEM into a widget you want, click on the “Add Variables” button, select your device and select the variable you want to be linked on that widget.



17. Then configure the widget as you like and click the green check mark. Some widgets can log data from more than one variable.

18. Now you have created a new widget which will show data from a variable you chose. **Note. You can always edit the created widgets afterwards.**

19. Next, you can keep on adding widgets for every variable you want to see on the dashboard. **The limit for widgets is 10, as is for the variables.**

Appendix 2. Holding Registers of the Microbrewery (in Finnish)

Metropolia Ammattikorkeakoulu PIENPANIMO

KÄYTTÖOHJE SIVU 1/3

Osoite	Skaala	Sisältö [yksikkö] (tehdasasetus)
A 1	1	Kattilan lämpötila TE1 [°C]
A 2	1	Kattilan lämpötilan ohjearvo [°C]
A 3	1	Menoveden lämpötila TE2 [°C]
A 4	1	Menoveden ohjearvo [°C]
A 5	1	Kuumavesisäiliön lämpötila TE3 [°C]
A 6	1	Kuumavesisäiliön ohjearvo [°C]
A 7	1	Lämpötila lämmönvaihtimen jälkeen TE4 [°C]
A 8	1	Lämmönvaihtimen asetus jäähdytyksessä (5.0 °C)
A 9	1	varalla
A 10	1	varalla
A 11	1	Käytössä olevan mäskäysportaan asetus [°C]
A 12	1	varalla
A 13	2	Konsentraatiomittaus [Brix] [0.00 - 30.00]
A 14	2	pH-mittaus [0.00 - 14.00]
A 15	2	varalla
A 16	2	varalla
A 17	2	Käymissäiliön 1 massa [kg]
A 18	2	Käymissäiliön 2 massa [kg]
A 19	2	Käymissäiliön 3 massa [kg]
A 20	2	Käymissäiliön 4 massa [kg]
A 21	1	Konsentraatiomittauksen lämpötila [°C]
A 22	1	pH-mittauksen lämpötila [°C]
A 23	1	Käymissäiliön 1 lämpötila TE21 [°C]
A 24	1	Käymissäiliön 2 lämpötila TE22 [°C]
A 25	1	Käymissäiliön 3 lämpötila TE23 [°C]
A 26	1	Käymissäiliön 4 lämpötila TE24 [°C]
A 27	1	Lohkoroottoripumpun ohjaus käsikäytöllä POT1 [%]
A 28	0	Kuumavesisäiliön lämmitysteho [kW]
A 29	1	Lämmitysventtiilin TV1 ohjausjännite [%]
A 30	1	Sekoittimen M1 ohjausjännite [%]
A 31	1	Lohkoroottoripumpun P2 ohjausjännite [%]
A 32	1	varalla
A 33	0	Sekoittimen ohjaustapa [0=seis, 1=käsin eteen, 2=käsin taakse, 3=automaatti]
A 34	0	Mitattu nousunopeus [sek/°C]
A 35	0	Ohjelmavaihe [0=seis, 1=lämmitys, 2=nosto, 3=porras, 4=keitto, 5=valmis, 6=keskeytys]
A 36	0	Mäskäysporras [1 - 5, 0=ei portaalla]
A 37	2	Portaan aikaa jäljellä [min.sek]
A 38	2	Aika prosessin alusta [hh.min]
A 39	0	Talletettavan reseptin numero [0 - 20]

A 40	0	Käytössä olevan reseptin numero [1 - 20]	
A 41	1	Porras 1, lämpötila-asetus [°C]	
	MÄSKÄYSRESEPTI		
A 42	0	Porras 1, pito-aika [min]	(A 41 - A 62)
A 43	1	Porras 1, sekoittimen ohjaus [%]	
...			

Metropolia Ammattikorkeakoulu PIENPANIMO

KÄYTTÖOHJE SIVU 2/3

Osoite	Skaala	Sisältö [yksikkö] (tehdasasetus)
A 53	1	Porras 5, lämpötila-asetus [°C]
A 54	0	Porras 5, pito-aika [min]
A 55	1	Porras 5, sekoittimen ohjaus [%]
A 56	0	Lämpötilan nostonopeus [60 sek/°C]
A 57	1	Sekoittimen ohjaus nostovaiheessa [50.0 %]
A 58	1	Maksimi lämpötilaero TE2 - TE1 nostossa (13.0 °C)
A 59	1	Minimi lämpötilaero TE2 - TE1 nostossa (8.0 °C)
A 60	1	Maksimi lämpötilaero TE2 - TE1 portaalla (3.0 °C)
A 61	1	Lämpötilan rajoitusalue ennen porrasta (1.0 °C)
A 62	1	Portaan lämpötilan hyväksymishystereesi (0.3 °C)
A 63	0	Portaan lämmityksen kompensoinnin jyrkkyys (10)
A 64	1	Mäskäyksen lämpötilan hälytyshystereesi (1.0 °C)
A 65	0	Mäskäyksen lämpötilan hälytysviive (10 sek)
A 66	0	Keittoaika [60 min]
A 67	1	Lämmitysveden asetus keitossa (115.0 °C)
A 68	1	Keittoajan aloituslämpötila (99.0 °C)
A 69	1	varalla
A 70	1	Sekoittimen ohjaus nostossa (50.0%)
A 71	1	Sekoittimen ohjaus keitossa (50.0%)
A 72	1	Lämmityssäiliön minimilämpötila (60.0 °C)
A 73	1	Lämmityssäiliön minimiero mäskäyksessä (20.0 °C)
A 74	1	Lämmityssäiliön lämpötila keitossa (120.0 °C)
A 75	1	Lämmityssäiliön säätöhystereesi (0.5 °C)
A 76	1	Lämmityssäiliön säätöviive (10 sek)
A 77	0	Keiton hälytysaika 1 (20 min) (nämä ei vielä ohjelmassa)
A 78	0	Keiton hälytysaika 2 (40 min)
A 79	0	Keiton hälytysaika 3 (60 min)
A 80	0	varalla
A 81	0	Käymissäiliön 1 tila [0 = ei käytössä, 1 = pito, 2 = alkujäähdytys]
A 82	1	Lämpötila-asetus (15.0 °C)
A 83	1	Lämpötilasäädön hystereesi (0.5 °C)
A 84	0	Lämpötilan säätöviive (10 sek)
A 85	1	Lämpötilan hälytyshystereesi (1.0 °C)
A 86	0	Lämpötilan hälytysviive (10 min)

A 87	0	Käymissäiliön 2 tila [0 = ei käytössä, 1 = pito, 2 = alkujäähdytys]
A 88	1	Lämpötila-asetus (15.0 °C)
A 89	1	Lämpötilasäädön hystereesi (0.5 °C)
A 90	0	Lämpötilan säätöviive (10 sek)
A 91	1	Lämpötilan hälytyshystereesi (1.0 °C)
A 92	0	Lämpötilan hälytysviive (10 min)
A 93	0	Käymissäiliön 3 tila [0 = ei käytössä, 1 = pito, 2 = alkujäähdytys]
A 94	1	Lämpötila-asetus (15.0 °C)
A 95	1	Lämpötilasäädön hystereesi (0.5 °C)
A 96	0	Lämpötilan säätöviive (10 sek)
A 97	1	Lämpötilan hälytyshystereesi (1.0 °C)
A 98	0	Lämpötilan hälytysviive (10 min)

Metropolia Ammattikorkeakoulu PIENPANIMO

KÄYTTÖOHJE SIVU 3/3

Osoite	Skaala	Sisältö [yksikkö] (tehdasasetus)
A 99	0	Käymissäiliön 4 tila [0 = ei käytössä, 1 = pito, 2 = alkujäähdytys]
A 100	1	Lämpötila-asetus (15.0 °C)
A 101	1	Lämpötilasäädön hystereesi (0.5 °C)
A 102	0	Lämpötilan säätöviive (10 sek)
A 103	1	Lämpötilan hälytyshystereesi (1.0 °C)
A 104	0	Lämpötilan hälytysviive (10 min)
...		
A 107	1	Sekoittimen miniminopeus (5.0 %)
A 108	1	Sekoittimen maksiminopeus (100.0 %)
A 109	1	Lohkoroottoripumpun miniminopeus (5.0 %)
A 110	1	Lohkoroottoripumpun maksiminopeus (100.0 %)
…		
A 112	0	Säätäjän numero [1=TV1, 2=LR-pumppu]
A 113	1	Säätöpoikkeama
A 114	1	Säädön ohjausarvo [%]
A 115	1	Säätöpoikkeaman minimi (-15.0)
A 116	1	Säätöpoikkeaman maksimi (15.0)
A 117	0	Säädön laskentaväli (3 sek)
A 118	0	I-säädön laskentaväli (1 x A 117)
A 119	0	I-säädön estoaika (5 sek)
A 120	0	Vahvistus (10)
A 121	0	I-säädön vahvistus (20)
F 1	Hälytysnäyttö	

HÄLYTYSKOODIT:

1 = mäskäyksen lämpötilapoikkeama tai TE1 anturivika
2 = lämmitysveden TE2 anturivika
3 = lämmityssäiliön lämpötilapoikkeama tai TE3 anturivika
4 = lämmönvaihtimen TE4 anturivika
5 = käymissäiliö 1 lämpötilapoikkeama tai TE21 anturivika
6 = käymissäiliö 2 lämpötilapoikkeama tai TE22 anturivika
7 = käymissäiliö 3 lämpötilapoikkeama tai TE23 anturivika

8 = käymissäiliö 4 lämpötilapoikkeama tai TE24 anturivika
9 = pintakytkimen LSE1 hälytys (lämmityssäiliö)
10 = taajuusmuuttajan SC1 hälytys (sekoitin)
11 = taajuusmuuttajan SC2 hälytys (lohkoroottoripumppu)

PLC Number = 1, 9600 Baud, NO parity, 1 stop bit.
MODBUS osoite = 42149 (tai 4X:2149) + A XX
eli A 1 = 42150 ja A 110 = 42259

Skaalaukset: Autologissa wordit tyyppiä UINT
0 = paljas luku
1 = 1 desimaali, offset -100.0 esim. lämpötilat -50.0 (500) - 154.7 (2547)
2 = 2 desimaalia, ei offsettiä

INDIKOINNIT: 42148

bit 0	sekoitin käy
bit 1	lohkoroottoripumppu käy vasemmalle
bit 2	lohkoroottoripumppu käy oikealle
bit 3	kiertovesipumppu päällä (ohjaus)
bit 4	pintakytkin LSE1 ok
bit 5	lämmitysvaraaja lämmin
bit 6	varalla
bit 7	reseptiä muutettu
bit 8	käymissäiliö 1 venttiili auki
bit 9	käymissäiliö 2 venttiili auki
bit 10	käymissäiliö 3 venttiili auki
bit 11	käymissäiliö 4 venttiili auki
bit 12	
bit 13	
bit 14	
bit 15	

HÄLYTYKSET: 42147

bit 0	mäskäyksen lämpötilapoikkeama tai TE1 anturivika
bit 1	lämmitysveden TE2 anturivika
bit 2	lämmityssäiliön lämpötilapoikkeama tai TE3 anturivika
bit 3	lämmönvaihtimen TE4 anturivika
bit 4	käymissäiliö 1 lämpötilapoikkeama tai TE21 anturivika
bit 5	käymissäiliö 2 lämpötilapoikkeama tai TE22 anturivika

bit 6	käymissäiliö 3 lämpötilapoikkeama tai TE23 anturivika
bit 7	käymissäiliö 4 lämpötilapoikkeama tai TE24 anturivika
bit 8	pintakytkimen LSE1 hälytys (lämmityssäiliö)
bit 9	taajuusmuuttajan SC1 hälytys (sekoitin)
bit 10	taajuusmuuttajan SC2 hälytys (lohkoroottoripumppu)

Appendix 3. Arduino IDE Code

The include, defines and global variables

```
#include <Controllino.h>
#include "ModbusRtu.h"
#include <Ethernet.h>
#include <SPI.h>
#include <UbidotsEthernet.h>

// This MACRO defines Modbus master address.
// For any Modbus slave devices are reserved addresses in the range from 1 to 247.
// Important note only address 0 is reserved for a Modbus master device!
#define MasterModbusAdd  0
#define SlaveModbusAdd  1
//SET COM PORT
#define RS485Serial     3
//Setup as ModbusMaster
Modbus ControllinoModbusMaster(MasterModbusAdd, RS485Serial, 0);

// ubidots defines
#define TOKEN "BBFF-gS2Ytzst5k1BbADFHFWyIZE3d9v215" // Assign your Ubidots TOKEN
#define DEVICE_LABEL "controllino"  // Put here your Ubidots variable ID

/* Enter a MAC address for your controller below */
byte mac[] = { 0x54, 0x83, 0x3A, 0x07, 0x2E, 0xB1 };

/* initialize the instance */
Ubidots client(TOKEN);
```

```
// This is an structe which contains a query to an slave device
modbus_t ModbusQuery[5];
//Define global variables
uint16_t ModbusSlaveRegisters1[29];
uint16_t ModbusSlaveRegisters2[29];
uint16_t ModbusSlaveRegisters3[29];
uint16_t ModbusSlaveRegisters4[29];
uint16_t ModbusSlaveRegisters5[5];

uint8_t myState; // machine state
uint8_t currentQuery; // pointer to message query
unsigned long WaitingTime;

int i = 1, j = 0;
int tankkiRegister[121];
```

The setup

```
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  Serial.print(F("Starting ethernet..."));
  if (!Ethernet.begin(mac)) {
    Serial.println(F("failed"));
  } else {
    Serial.println(Ethernet.localIP());
  }
  client.setDeviceLabel("controllino");
  /* Give the Ethernet shield a second to initialize */
  delay(2000);
  Serial.println(F("Ready"));
  delay(1000);
```

```
  Serial.println("----------------------------------------");
  Serial.println("CONTROLLINO Modbus RTU Master Test Sketch");
  Serial.println("----------------------------------------");
  Serial.println("");
  // ModbusQuery 0: read registers
  ModbusQuery[0].u8id = SlaveModbusAdd; // slave address
  ModbusQuery[0].u8fct = 3; // function code (this one is registers read)
  ModbusQuery[0].u16RegAdd = 2149; // start address in slave
  ModbusQuery[0].u16CoilsNo = 29; // number of elements (coils or registers) to
read
  ModbusQuery[0].au16reg = ModbusSlaveRegisters1; // pointer to a memory
array in the CONTROLLINO

  // ModbusQuery 1: read registers
  ModbusQuery[1].u8id = SlaveModbusAdd; // slave address
  ModbusQuery[1].u8fct = 3; // function code (this one is registers read)
  ModbusQuery[1].u16RegAdd = 2178; // start address in slave
  ModbusQuery[1].u16CoilsNo = 29; // number of elements (coils or registers) to
read
  ModbusQuery[1].au16reg = ModbusSlaveRegisters2; // pointer to a memory
array in the CONTROLLINO

  // ModbusQuery 2: read registers
  ModbusQuery[2].u8id = SlaveModbusAdd; // slave address
  ModbusQuery[2].u8fct = 3; // function code (this one is registers read)
  ModbusQuery[2].u16RegAdd = 2207; // start address in slave
  ModbusQuery[2].u16CoilsNo = 29; // number of elements (coils or registers) to
read
  ModbusQuery[2].au16reg = ModbusSlaveRegisters3; // pointer to a memory
array in the CONTROLLINO

  // ModbusQuery 3: read registers
```

```
  ModbusQuery[3].u8id = SlaveModbusAdd; // slave address
  ModbusQuery[3].u8fct = 3; // function code (this one is registers read)
  ModbusQuery[3].u16RegAdd = 2236; // start address in slave
  ModbusQuery[3].u16CoilsNo = 29; // number of elements (coils or registers) to
read
  ModbusQuery[3].au16reg = ModbusSlaveRegisters4; // pointer to a memory
array in the CONTROLLINO

  // ModbusQuery 4: read registers
  ModbusQuery[4].u8id = SlaveModbusAdd; // slave address
  ModbusQuery[4].u8fct = 3; // function code (this one is registers read)
  ModbusQuery[4].u16RegAdd = 2265; // start address in slave
  ModbusQuery[4].u16CoilsNo = 5; // number of elements (coils or registers) to
read
  ModbusQuery[4].au16reg = ModbusSlaveRegisters5; // pointer to a memory
array in the CONTROLLINO



  ControllinoModbusMaster.begin( 9600 ); // baud-rate at 19200
  ControllinoModbusMaster.setTimeOut( 5000 ); // if there is no answer in 5000
ms, roll over

  WaitingTime = millis() + 3000;
  myState = 0;
  currentQuery = 0;
}
```

The loop

```
void loop() {
  Ethernet.maintain();
  switch ( myState ) {
    case 0:
      if (millis() > WaitingTime) myState++; // wait state
      break;
    case 1:
      Serial.print("---- Sending query ");
      Serial.print(currentQuery);
      Serial.println(" ----------");
      ControllinoModbusMaster.query( ModbusQuery[currentQuery] ); // send
query (only once)
      myState++;
      currentQuery++;
      if (currentQuery == 6)
      {
        currentQuery = 0;
      }
      break;
    case 2:
      ControllinoModbusMaster.poll(); // check incoming messages
      if (ControllinoModbusMaster.getState() == COM_IDLE)
      {
        // response from the slave was received
        myState = 0;
        WaitingTime = millis() + 2000;
        // debug printout
        if (currentQuery == 1)
        {
          i = ModbusQuery[0].u16RegAdd;
```

```
        j = 0;
        // registers read was proceed
        Serial.println("---------- READ RESPONSE RECEIVED ----");
        Serial.print("Slave ");
        Serial.println(SlaveModbusAdd, DEC);



        for (i = ModbusQuery[currentQuery - 1].u16RegAdd ; i <
ModbusQuery[currentQuery].u16RegAdd; i++) {
          Serial.print("REGISTER ");
          Serial.print(j + 1);
          Serial.print(": ");
          Serial.println(ModbusSlaveRegisters1[j]);
          tankkiRegister[j] = ModbusSlaveRegisters1[j];
          j++;
        }
      }
      else if (currentQuery == 2) {
        int k = 0;
        for (i = ModbusQuery[1].u16RegAdd ; i < ModbusQuery[2].u16RegAdd;
i++) {
          Serial.print("REGISTER ");
          Serial.print(j + 1);
          Serial.print(": ");
          Serial.println(ModbusSlaveRegisters2[k]);
          tankkiRegister[j] = ModbusSlaveRegisters2[k];
          j++;
          k++;
        }
      }
      else if (currentQuery == 3) {
```

```
        int k = 0;
        for (i = ModbusQuery[2].u16RegAdd ; i < ModbusQuery[3].u16RegAdd;
i++) {
          Serial.print("REGISTER ");
          Serial.print(j + 1);
          Serial.print(": ");
          Serial.println(ModbusSlaveRegisters3[k]);
          tankkiRegister[j] = ModbusSlaveRegisters3[k];
          j++;
          k++;
        }
      }
      else if (currentQuery == 4) {
        int k = 0;
        for (i = ModbusQuery[3].u16RegAdd ; i < ModbusQuery[4].u16RegAdd;
i++) {
          Serial.print("REGISTER ");
          Serial.print(j + 1);
          Serial.print(": ");
          Serial.println(ModbusSlaveRegisters4[k]);
          tankkiRegister[j] = ModbusSlaveRegisters4[k];
          j++;
          k++;
        }
      }
      else if (currentQuery == 5) {
        int k = 0;
        for (i = ModbusQuery[4].u16RegAdd ; i < 2270; i++) {
          Serial.print("REGISTER ");
          Serial.print(j + 1);
          Serial.print(": ");
          Serial.println(ModbusSlaveRegisters5[k]);
```

```
          tankkiRegister[j] = ModbusSlaveRegisters5[k];
          j++;
          k++;
        }
        SendToUbidots();
      }
    }
    break;
  }
}
```

Send to Ubidots

```
//Define the desired variable labels for Ubidots
//Maximum of 10 variables, comment out the labels that are not needed
//These are the names you will see in the Ubidots platform as variables
#define VARIABLE_LABEL1 "BRIX"
#define VARIABLE_LABEL2 "pH"
#define VARIABLE_LABEL3 "state"
#define VARIABLE_LABEL4 "kettletemp"
#define VARIABLE_LABEL5 "flushwatertemp"
#define VARIABLE_LABEL6 "tanktemp1"
#define VARIABLE_LABEL7 "tanktemp2"
#define VARIABLE_LABEL8 "tanktemp3"
#define VARIABLE_LABEL9 "tanktemp4"
//#define VARIABLE_LABEL10 "xxxx"

/* examples =
 *  You want to add the variable: temperature of pH measure to Ubidots
 *  define a Variable label not in use with a name suited for it
 *  #define VARIABLE_LABEL10 "pHtempMeasure"
 */
```

```
void SendToUbidots() {
  float BRIX = scaling2(tankkiRegister[12]);              //Array tankkiRegister[] contains data from all of TANKKI's holding registers
  float pH = scaling2(tankkiRegister[13]);                //Check from Tankkis register map to see which register contains which value
  float kettletemp = scaling1(tankkiRegister[0]);         //tankkiRegister[0] = Holding register 42150
  float flushwatertemp = scaling1(tankkiRegister[2]);     //tankkiRegister[x] = Holding register x + 42150
  int state = tankkiRegister[86];                         //Define your variables and check the scaling if needed in Tankkis register map
  float tanktemp[4];                                      //functions are scaling1(tankkiRegister[x]); and scaling2(tankkiRegister[x]);
  tanktemp[0] = scaling1(tankkiRegister[22]);             //Use float or double values if decimals are needed
  tanktemp[1] = scaling1(tankkiRegister[23]);
  tanktemp[2] = scaling1(tankkiRegister[24]);
  tanktemp[3] = scaling1(tankkiRegister[25]);

/* examples =
 * You want to add the value of the temperature of pH measure to Ubidots
 * Check the holding register number = A22 = 42171 --> tankkiRegister[x] = x+42150 --> tankkiRegister[21]
 * Then check if scaling is needed --> Holding register 42171 uses scaling number 1 --> scaling1(tankkiRegister[21]
 * Then choose a variable name suited for the data contained in the register --> pHtemp
 * Define the variable as a floating point and give it the corresponding value of holding register 42171 with scaling number 1
 * float pHtemp = scaling1(tankkiRegister[21];
 */
```

Appendix 3

```
/*_____
_____
_____*/
  client.add(VARIABLE_LABEL1, BRIX);                    //Variable label =
Variable name you want in ubidots
  client.add(VARIABLE_LABEL2, pH);                     //Second parameter =
variable in code (can be anything)
  client.add(VARIABLE_LABEL3, state);                  //Variables have to be
sent maximum 5 at a time
  client.add(VARIABLE_LABEL4, kettletemp);               //uncomment or
comment the variables needed or not needed
  client.add(VARIABLE_LABEL5, flushwatertemp);
  client.sendAll();

  client.add(VARIABLE_LABEL6, tanktemp[0]);
  client.add(VARIABLE_LABEL7, tanktemp[1]);
  client.add(VARIABLE_LABEL8, tanktemp[2]);
  client.add(VARIABLE_LABEL9, tanktemp[3]);
  //client.add(VARIABLE_LABEL10, yourVariable);
  client.sendAll();
}
/* examples =
 * you want to send the value of the temperature of pH measure to Ubidots
 * in previous examples we have defined the VARIABLE_LABEL10 as
pHtempMeasure
 * We have also given the variable phtemp the value of the holding register
42171 with number 1 scaling
 * Now all we have to do is write or modify an existing variable:
 * client.add(VARIABLE_LABEL10, phtemp);
 */
```

The scaling functions

```
float scaling1(int tankkiRegister) {
  float v = tankkiRegister;
  v -= 1000;
  v /= 10;
  return v;
}
float scaling2(int tankkiRegister) {
  float v = tankkiRegister;
  v /= 100;
  return v;
}
```

Appendix 4

4. Pictures of the project