SAVONIA

Serial communication & MODBUS protocol implementation using .NET framework

Trung-Hieu Nguyen

Bachelor's Thesis

___. ___. _____ ______________________________

Bachelor's degree (UAS)

SAVONIA UNIVERSITY OF APPLIED SCIENCES **THESIS Abstract**

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Trung-Hieu Nguyen			
Title of Thesis Serial communication and MODBUS protocol implementation using .NET framework			
Date	25.05.2014	Pages/Appendices	46/5
Supervisor(s) Arto Toppinen			
Client Organisation/Partners Savonia University of Applied Sciences			

Abstract
In English:
The thesis introduces serial communication used in information technology and some common serial ports used in industry. Then, it presents the project work which involves developing a data acquisition software using .NET framework to collect electrical measurements of the whole Savonia UAS's power consumption. The communication between the software and ABB M2M measuring device uses MODBUS protocol over RS-485 serial port. After collecting measurements, the software sends them to database server using WCF (Windows Communication Foundation) technology.

Suomeksi:
Opinnäytetyö esittelee tietotekniikassa käytettyä sarjaliikennettä sekä joitain teollisuudessa yleisesti käytettyjä sarjaportteja. Se käsittelee myös projektityötä jossa kehitettiin .NET frameworkiin pohjautuva tiedonkeruujärjestelmä joka kerää tietoa Savonia AMK:n sähkönkulutuksesta. Ohjelman ja ABB M2M mittalaitteen välillä käytetään MODBUS protokollaa RS-485 sarjaportin yli. Mittausten keruun jälkeen ohjelmisto lähettää tiedot serverille tietokantaan käyttäen WCF (Windows Communication Foundation) tekniikkaa.

Keywords
Serial Communication, MODBUS, .NET framework, RS-232, RS-485, USB, C#, Visual Studio, ABB, M2M

Abbreviations:

USB	Universal Serial Bus
bps	bits per second
Kbps	Kilobits per second
Mbps	Megabits per second
MSB	Most Significant Bit
LSB	Least Significant Bit
Tx	Transmit
Rx	Receive
CTS	Clear-To-Send
RTS	Request-To-Send
DTR	Data Terminal Ready
DSR	Data Set Ready
CRC	Cyclic Redundancy Check
Hex	Hexadecimal
UPS	Uninterpretable Power Supply
DTE	Data Terminal Equipment
DCE	Data Communication Equipment
SE0	Single-ended 0
EOP	End-Of-Package
TCP	Transmission Control Protocol
RTU	Remote Terminal Unit
ASCII	American Standard Code for Information Interchange
LRC	Longitudinal Redundancy Check
CRC	Cyclical Redundancy Check
xxh	Number in Hexadecimal
xxd	Number in Decimal
word	2-byte
IDE	Integrated Development Environment
API	Application Programming Interface
WCF	Windows Communication Foundation

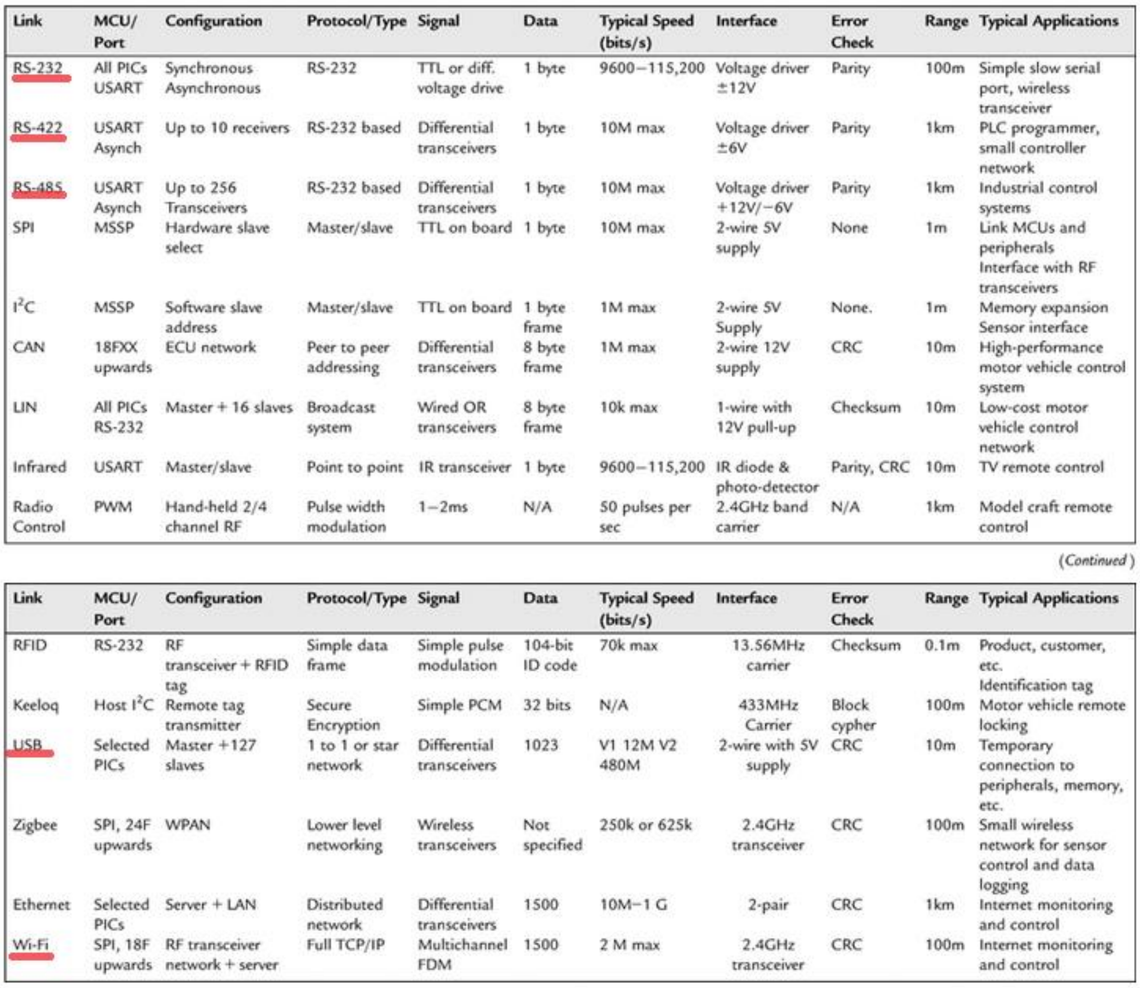
CONTENTS

1 Introduction

Serial Communication is not a new technology. Smoke signal communication had been used by people since ancient time. Morse code was developed and used since 1800s for telegraph messaging. ASCII was introduced since 1960s. However, since the emergence of computer era, serial communication is still used in telecom and computer science today and serial ports appear as external connections in almost all electronics devices such as computers, smartphones, and digital cameras. Knowledge about serial communication in Information Technology is still relevant. Therefore, in first part of the thesis, I wrote a study on serial communication used in computer technology and introduce some common serial ports used in industry. In the second part, I introduce my final project work with Savonia University of Applied Sciences, which is the development of data acquisition software written in C#, using .NET framework to collect data from electrical measuring box in real-time and send them to database server. The communication between PC and electrical measuring box uses MODBUS protocol over RS485, a serial port standard.

1.1 Serial Communication

In computer technology, serial communication is a process of transmitting one bit at a time between devices. This is the most common method to communicate between embedded system and micro-controller as well as between devices in computer network. Sending one bit at a time seems to be primitive and inefficient but it has its own advantage. The simplicity enables to use small cheap cables and connectors. In contrast, parallel communication transfers multiple bits a time over multiple wires. This method is faster but less flexible and needs bigger cables. One other major advantage of serial communication is that the range can be very long. RS-232 cable length can be up to 100 meters long, and RS-485 cable length can be up to 1000 meters long [1].



Link	MCU/ Port	Configuration	Protocol/Type	Signal	Data	Typical Speed (bits/s)	Interface	Error Check	Range	Typical Applications
RS-232	All PICs USART	Synchronous Asynchronous	RS-232	TTL or diff. voltage drive	1 byte	9600–115,200	Voltage driver ±12V	Parity	100m	Simple slow serial port, wireless transceiver
RS-422	USART Asynch	Up to 10 receivers	RS-232 based	Differential transceivers	1 byte	10M max	Voltage driver ±6V	Parity	1km	PLC programmer, small controller network
RS-485	USART Asynch	Up to 256 Transceivers	RS-232 based	Differential transceivers	1 byte	10M max	Voltage driver +12V/−6V	Parity	1km	Industrial control systems
SPI	MSSP	Hardware slave select	Master/slave	TTL on board	1 byte	10M max	2-wire 5V supply	None	1m	Link MCUs and peripherals Interface with RF transceivers
I²C	MSSP	Software slave address	Master/slave	TTL on board	1 byte frame	1M max	2-wire 5V Supply	None.	1m	Memory expansion Sensor interface
CAN	18FXX upwards	ECU network	Peer to peer addressing	Differential transceivers	8 byte frame	1M max	2-wire 12V supply	CRC	10m	High-performance motor vehicle control system
LIN	All PICs RS-232	Master + 16 slaves	Broadcast system	Wired OR transceivers	8 byte frame	10k max	1-wire with 12V pull-up	Checksum	10m	Low-cost motor vehicle control network
Infrared	USART	Master/slave	Point to point	IR transceiver	1 byte	9600–115,200	IR diode & photo-detector	Parity, CRC	10m	TV remote control
Radio Control	PWM	Hand-held 2/4 channel RF	Pulse width modulation	1–2ms	N/A	50 pulses per sec	2.4GHz band carrier	N/A	1km	Model craft remote control
RFID	RS-232	RF transceiver + RFID tag	Simple data frame	Simple pulse modulation	104-bit ID code	70k max	13.56MHz carrier	Checksum	0.1m	Product, customer, etc. Identification tag
Keeloq	Host I²C	Remote tag transmitter	Secure Encryption	Simple PCM	32 bits	N/A	433MHz Carrier	Block cypher	100m	Motor vehicle remote locking
USB	Selected PICs	Master +127 slaves	1 to 1 or star network	Differential transceivers	1023	V1 12M V2 480M	2-wire with 5V supply	CRC	10m	Temporary connection to peripherals, memory, etc.
Zigbee	SPI, 24F upwards	WPAN	Lower level networking	Wireless transceivers	Not specified	250k or 625k	2.4GHz transceiver	CRC	100m	Small wireless network for sensor control and data logging
Ethernet	Selected PICs	Server + LAN	Distributed network	Differential transceivers	1500	10M–1 G	2-pair	CRC	1km	Internet monitoring and control
Wi-Fi	SPI, 18F upwards	RF transceiver network + server	Full TCP/IP	Multichannel FDM	1500	2 M max	2.4GHz transceiver	CRC	100m	Internet monitoring and control

Fig. 1 Characteristics of some common serial ports [1].

Some protocols are discussed in this report: RS-232 with the range up to 100 m, RS-422 & RS-285 with the range up to 1 km, USB with the range up to 10 m, Wi-Fi (wireless) with the range up to 100 m.

Serial Communication also has some drawbacks.

- There is no single universal interface for all kind of purpose. Depending on requirement of the communication, suitable protocols and ports are used.
- In addition, maximum bit rate of RS-232 is 20 Kbps [2 chapter 1], but there are several faster interface. RS-485 can be up to 10 Mbps [1]. PC to USB Virtual COM (USB 2.0) can be up to 480M. [1].
- In communication between PC (Windows for example) and embedded system using Serial port, the real-time performance is not guaranteed due to multitask management of the system. But nowadays, the speed of PC micro-processor is faster and the delay of communication is smaller. Therefore, in practice, the latency is not a matter. In embedded system like Arduino, the schedule of communication can be controlled more accurate. [2 chapter 1]

1.2 Characteristics

1.2.1 Asynchronous and Synchronous Communications

1.2.1.1 Synchronous Communication

In communication between two devices, at a time, one is the sender and the other is the receiver. In bidirectional communication, typically, there are 2 lines corresponding for Rx-Tx (Receive-Transmit) and Tx-Rx (Transmit-Receive). In synchronous transmission, an external clock is used as timing reference for both transmitter and receiver. This clock is controlled by either of two ends. Each bit is available for a predefined clock time. Transmitting and receiving is triggered on falling and rising edge of the clock pulse depending on protocol. Examples of this method of synchronization interface are I2C, SPI, Microwire.

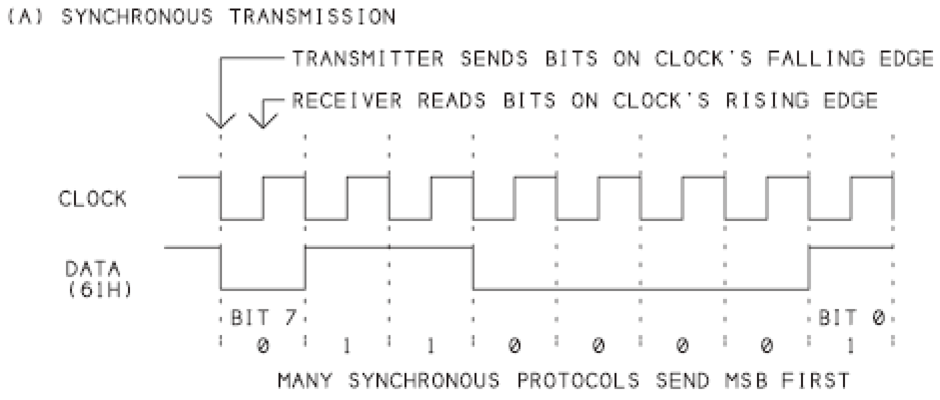

(A) SYNCHRONOUS TRANSMISSION
TRANSMITTER SENDS BITS ON CLOCK'S FALLING EDGE
RECEIVER READS BITS ON CLOCK'S RISING EDGE
CLOCK
DATA (61H)
BIT 7
BIT 0
0 1 1 0 0 0 0 1
MANY SYNCHRONOUS PROTOCOLS SEND MSB FIRST

Fig. 2 An example of synchronous communication [2 chapter 2]

The byte is transmitted with MSB first. Each bit is sent at the falling edge of clock pulse. The receiving end will check the value of the bit at the rising edge.

1.2.1.2 Asynchronous Communication

In contrast of synchronous communication, there is no mutual clock line. Each device has its own internal clock. Both ends must agree on one common bit rate and few percent of error in timing is accepted. The communication is started when a start bit from transmitter is sent. It will continue with its internal clock rate. On the other end, after detecting the starting bit, the receiver uses its internal clock rate to track the follow bits. Since they both use the same clock rate, the communication is synchronized.

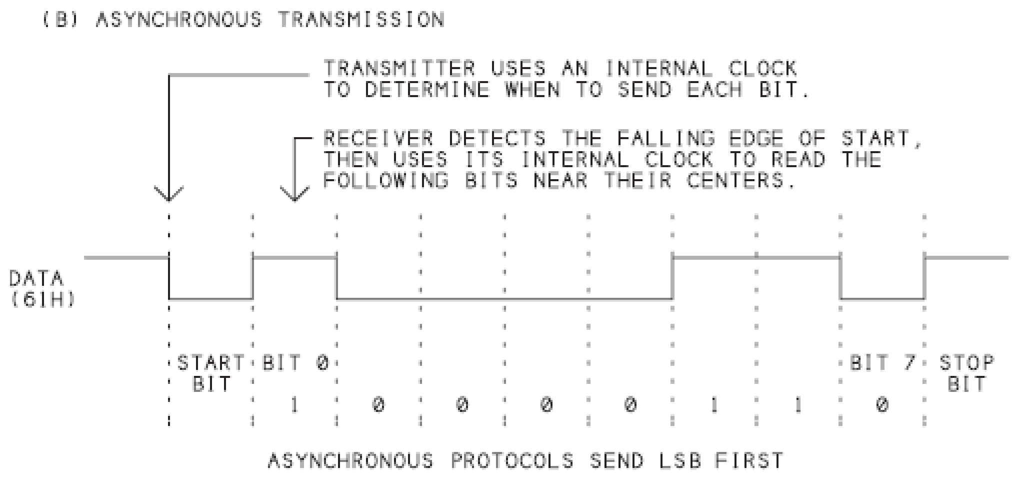

(B) ASYNCHRONOUS TRANSMISSION
TRANSMITTER USES AN INTERNAL CLOCK TO DETERMINE WHEN TO SEND EACH BIT.
RECEIVER DETECTS THE FALLING EDGE OF START, THEN USES ITS INTERNAL CLOCK TO READ THE FOLLOWING BITS NEAR THEIR CENTERS.
DATA (61H)
START BIT
BIT 0
1
0
0
0
0
1
1
BIT 7
0
STOP BIT
ASYNCHRONOUS PROTOCOLS SEND LSB FIRST

Fig. 3 An example of asynchronous communication [2 chapter 2]

The byte is transmitted with LSB first. The transmitter uses its own clock rate to send each bit. After detecting the start bit, the receiver uses its clock rate to follow.

1.2.2 Bit Rate and Baud rate

Bit rate is speed of transmitting or receiving of the device, which is measured by the number of bits per second can be transferred. Baud rate is the number of data unit can be transferred per second. In scope of the interfaces covered by this report, bit rate and baud rate are the same. However, in some communication such as phone lines and high-speed modems, methods like phase shifts to encode multiple bits in 1 data-transferring period, the bit rate is bigger than the baud rate.

The speed of transferring a character depends of how many bits it contains and how many overheads it needs. For example, an ASCII character has 7 bits in an asynchronous communication with 1 start bit and 1 stop bit. So the total number of bits for sending the characters is 9. The character sent rate will be bit rate dividing number of total bits.

Bit rate (bps)	Character rate (characters per second)
1200	133.3
2400	266.7
9600	1066.7

Table 1. Character rate of 7-bit word (1 stop bit) calculated from bit rate

1.2.3 Flow Control

In theory, a communication can work fine but in practice, there are many external factors that could lead to unsuccessful transmission or receiving, delays and missing bits. To prevent these kinds of problem, multiple techniques are used. Flow control (also called Handshaking) is one of these techniques. The idea of the method is simple. An additional line is used to indicate whether it is possible for transmitter to send the data. If the receiver is still busy processing the previous data or is not ready to receive yet, the transmitter will wait until the sending line is ready. For bidirectional communication, two indication lines will be used for two data transmission lines correspondingly. The flow control can be at either hardware level or software level. In RS-232 port, specific pins are dedicated for flow control called CTS (Clear-To-Send) and RTS (Request-To-Send). CTS is for input side or receiver to indicate that it is ready to receive data, while RTS is for output side or transmitter to indicate that it has data to send.

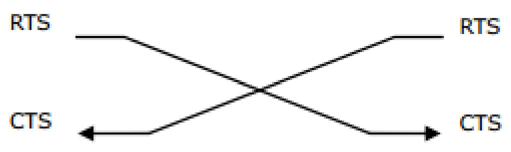


Fig. 4 RTS\CTS Flow control cable wiring in 2-way communication: one is sender but also receiver. [3 page 3]

In addition to CTS and RTS, there are two additional signals called DTR (Data Terminal Ready) and DSR (Data Set Ready). This is commonly used to establish the communication. But in some applications, they can be also used for flow control similar to RTS and CTS.

Furthermore, flow control can be implemented at software level. Receiver can send on Tx-Rx line a signal called Xon to indicate it is ready to receive data and Xoff to signal stopping seding. Xon is typically 0x11, while Xoff is 0x13. Thus, communication content must not contain these characters which are Ctrl+Q (Xon) and Ctrl+S (Xoff).

1.2.4 Error Checking

In addition to flow control, error checking is used to prevent wrong or incomplete data transmission which may be resulted from cable malfunctions or electrical interference. There are several methods in error checking such as adding parity bit, checksum and CRC (cyclic redundancy check).

1. In parity bit technique, additional bit will be added at the end of the sending data to indicate number of on bit (1-bit) odd or even. Option of odd or even parity bit is predefined and agreed by both sender and receiver. After receiving transmitted word, receiver will calculate parity bit again. If it doesn't match, error signal will be raised for resending or ignoring depending on application.

8-bit word	Parity bit (count of 1 bit)	Odd parity bit added in the end	Even parity bit added in the end
00110011	4	00110011**0**	00110011**1**
10111111	7	10111111**1**	10111111**0**
11111111	8	11111111**0**	11111111**1**
00101111	5	00101111**1**	00101111**0**

Table 2. Example of how parity bit is calculated and added to transferring word (the bit in bold format at the end).

2. In Checksum method, mathematical calculation such as counting length will be performed on data bits, then the result will be added to the block of sending data. Then calculation will be performed again on received data to compare.

3. CRC is a more sophisticated calculation using polynomial arithmetic to obtain the checksum value. As observing from two previous techniques, simple algorithms of calculating checksum can leave the error undetected. Strong checksum calculation function must ensure the odd of two different word matching each other small. The CRC algorithm resulting in 32-bit Hex number has the odds of two random numbers matching 1/ (2^32). This means the odds of two different words having the same CRC are roughly 1 in 4 billion. Furthermore, the result should be distributed evenly in its possible range, which means each bit of the data word has the same chance to change any bit of CRC result value.

Parity bit, in other words, is the special case of CRC technique where the calculated CRC register just has 1 bit.

1.3 Common Standards

1.3.1 RS232

1.3.1.1 Overview

RS-232 used to be standard for modem, UPS, typical peripherals such as mouse, keyboard, printer... RS stands for “Recommended Standard”. But nowadays it is not common anymore because of its limitation in transmission rate, big physical port size and high voltage usage. It is replaced by USB standard. However, RS-232 are still be used in industrial systems, scientific instruments or development devices because it is cheap and reliable for short distance and not-high-rate-required communication. With 2 lines of data transmission for 2 directions, RS-232 can perform full-duplex communication.

1.3.1.2 Line Voltage

The data is sent through data line, which are pulses of voltage referencing to ground line. Logic “1” is from -3V to -25V, typically -12V. Logic “0” is from +3V to 25V, typically +12V. Between -3V and +3V is considered undetermined-state [4 page 165].

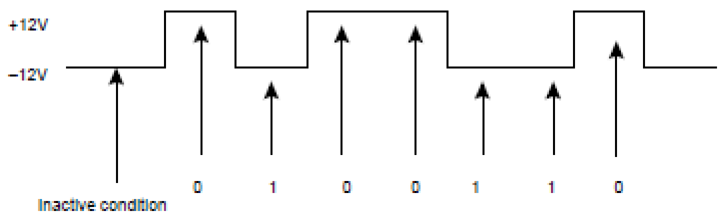


Fig.5 Typical voltage level of RS-232[4 page 166]

1.3.1.3 Connectors

The first typical type is DB-25 having D-shape 25 pins. RS-232 originally used this type of port, thus it has RS-232 full functionality. The DCE device (cable connector) has female pins and male outer case, while DTE device (computer outlet) has male pins and female outer case.

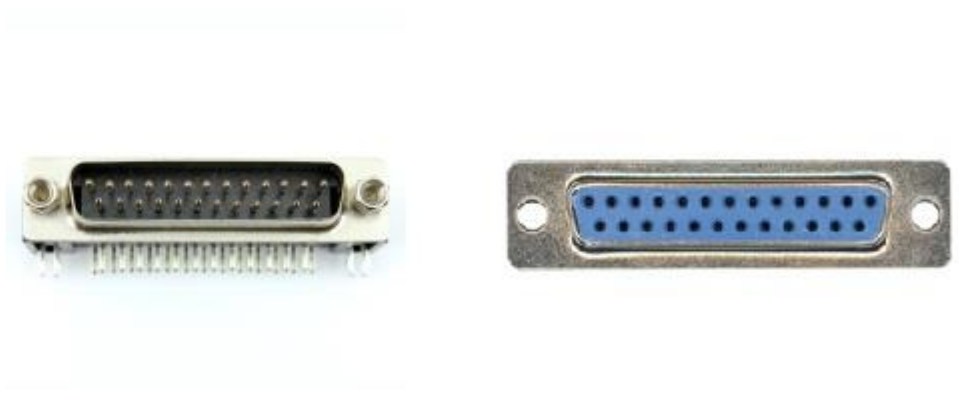
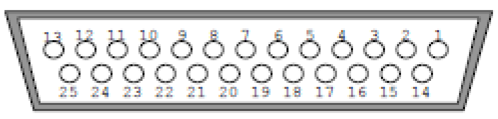


Fig.6 Example of male (left [5]) and female (right [6]) pin DB-25 connector

The functionality of primary pins is listed in the figure below. Some other pins play secondary role as backup for the main one. They works correspondingly the same. There are 3 main type of lines: Data, Control and Ground.



Pin	Signal
2	TxData
3	RxData
4	RTS
5	CTS
6	DSR
7	GND
20	DTR

Fig. 7 DB-25 Pin meaning-Date line: Tx is transmit, Rx is Receive. Control line: RTS: Request-To-Send, CTS: Clear-To-Send, DTR: Data-Terminal-Ready, DSR: Data-Set-Ready. Ground line: GRD: Ground [4 page 166]

Because of the large size of DB-25, it is not convenient. The DB-9 with D-shaped 9 pins was introduced. Those secondary pins are omitted resulting smaller in size. This is quite common implemented in personal computer quite some time ago (not anymore and be replaced by USB nowadays). Female and Male pins are assigned between DCE and DTE devices the same way as DB-25.

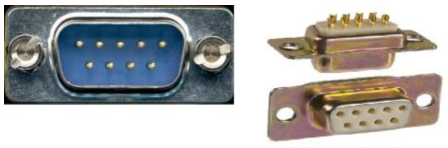
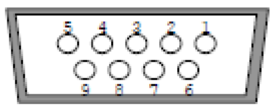


Fig.8 Example of commercial Male and Female DB-9 connectors [7]

Pin meanings are shown in the figured below.



Pin	Signal
2	RxData
3	TxData
4	DTR
5	GND
6	DSR
7	RTS
8	CTS

Fig.9 Pin functionality of DB-9 connector [4 page 166]

1.3.2 RS485

Instead of referencing to the ground like in RS-232, in RS-485 technology uses 2 wires to transmit data, and check the voltage difference between them to determine logic “1” or “0”. As the result, the bit rate can be faster than RS-232, up to 10Mbps [1] and the maximum range can be longer, up to 1km. Typically, there are 3 lines in RS485: A, B and C. A & B is data lines and C is ground. The signal is determined by the polarity between A and B. If A is negative to B, logic “1” is determined. On the other hand, B is negative to A, logic “0” is determined. Typically, the transmitter differential output is minimum 1.5V (up to 6.0V) and the receiver can detect at least 200mV [8]. Because of this range of detection, RS-485 is sufficient to be used in electromagnetic noisy environment like factories which may produce a lot of spikes to communication lines.

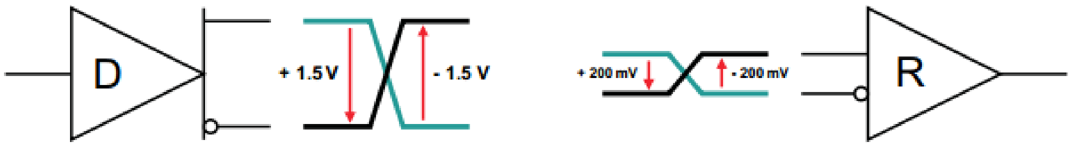


Fig. 10 RS-485 specified minimum signal levels [8 page 2]

Unlike RS-232 technology which is point-to-point communication, RS-485 allows multidrop network which has multiple transmitters and receivers connected to a common circuit. Terminating resistors (120 Ohm recommended [8]) must be connected in both ends of the common line to prevent data reflection.

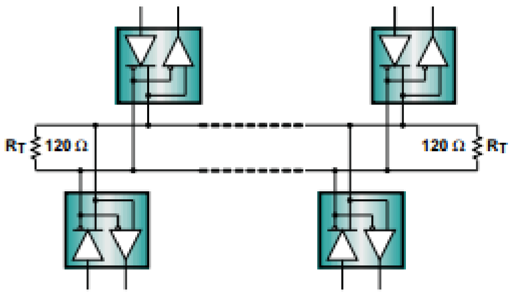


Fig. 11 RS-485 circuit with two terminating Resistors at both ends of common transmission cable [8 page 3]

With two-line common line circuit, RS-485 can only perform half-duplex communication. To implement full-duplex, four wires can be used to differentiate receive and transmit lines.

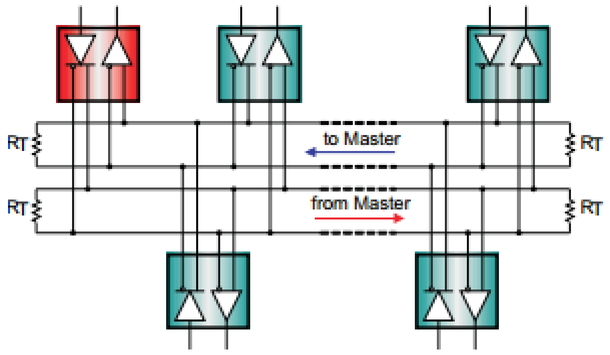


Fig. 12 Full-duplex RS-485 circuit implementation [8 page 2]

1.3.3 USB

1.3.3.1 Overview

USB, Universal Serial Bus is the most common serial port used today. It is implemented in almost all computers, scientific equipment, new televisions, smart-phones... USB has a lot of advantages.

Like its name, it is actually universal standard for PC peripherals. Nowadays, mass storage, mouse, printers, almost all use USB standard.

- Hot plug: Since Windows and other operating systems already store library of driver and they can download and install the driver if it is missing, the device using USB can be plug-and-play conveniently.
- No user setting: The end user don't have to set configurations such baud rate, stop bits... The host automatically recognizes and classifies connecting devices into their corresponding speed grade.
- Small cable connector: The size of USB connector is quite small, multiple ports can be implemented in one laptop conveniently. In smart-phones, micro USB is typically used. The size of the mounting can be down to 6.9mm x 1.86mm (inside intersection) for jack connector 6.9mm x 1.8mm [9 page 34].

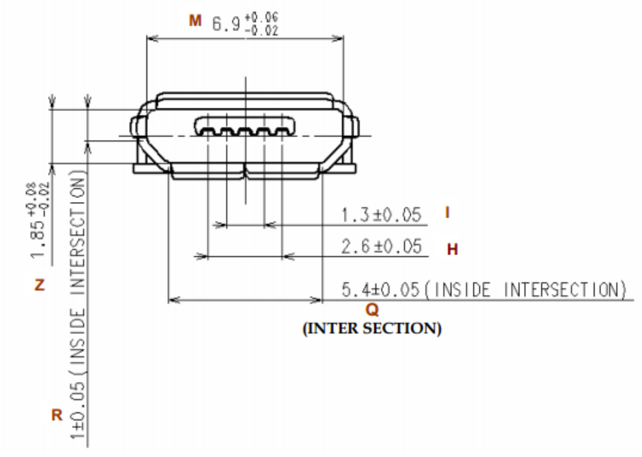


Fig. 13 Micro-USB B-Receptacle physical schematic [9 page 35]

- Multiple speed grades are used to maximize communication capacity and which one to be used is determined automatically by circuits and drivers. This is useful since lower-speed device is cheaper to produce. Thus the standard is manufacturer-friendly.

Interface	Max distance	Speed	Typical Use
USB 2.0	4.8 meters (up to 21 meters with 5 hubs)	Low speed: 1.5Mbps Full speed: 12 Mbps High speed: 480 Mbps	Peripherals, drive, speaker, printer...
USB 3.0	2.7 meters (up to 15 meters with 5 hubs)	Above speeds and Superspeed: 5Gbps	Mass storage, video

Table 3. USB versions comparison [11 chapter 1 page 3]

1.3.3.2 Topology

In USB technology, there will be always one host system, which is Host Controller (which is typically the computer). The communication using polled method. The host controller starts all transfers. Up to 127 peripherals can be connected to Host Controller or Root Hub. Multiple devices can be connected into a hub. The hub serves as USB device to its host and serves as the host for USB devices connected to it. Up to 5 external hubs can be used in topology.

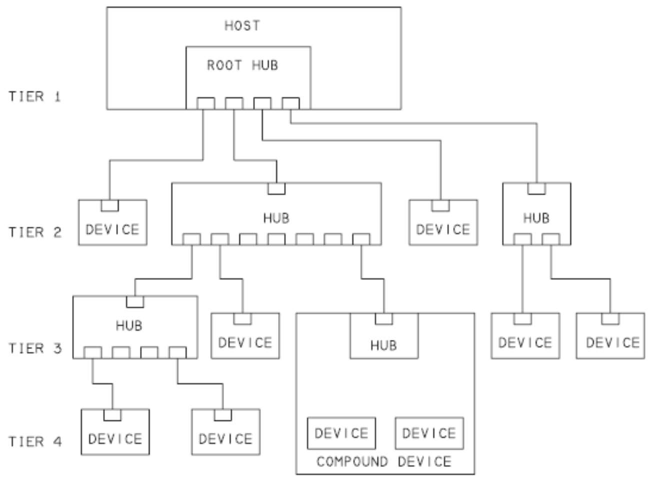


Fig. 14 Example of USB topology [11 chapter 1 page 16]

1.3.3.3 Data transfers

There are 4 types of data transfers which is used for different purposes and in different types of USB devices.

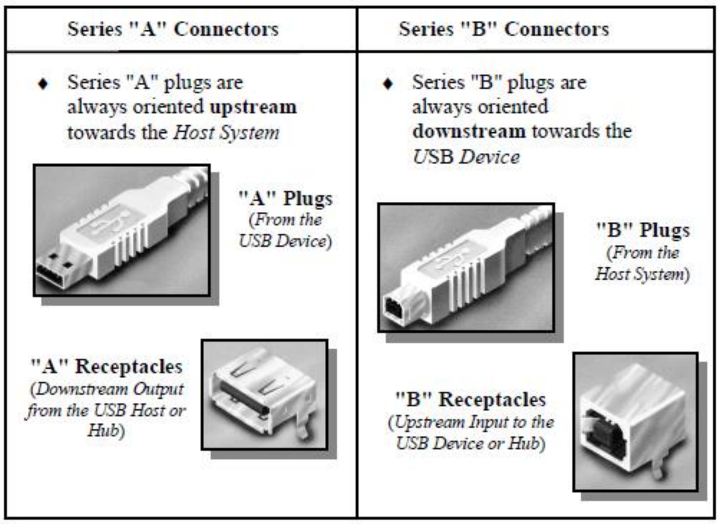
1. **Control Transfer:** When the USB device is connected to host, this kind of transfer is used for configure settings software on host controller.
2. **Bulk Transfer:** This is used for large data transferring such as printer. The communication must be loss-less while real-time speed is not required.

Reliability is ensured by error checking. Bandwidth allocation is not fixed, depending on activities

3. **Interrupt Transfer:** This is used for peripherals like mouse and keyboards where latency must be as low as possible. An event notification mechanism is used instead of constant bandwidth allocation.
4. **Isochronous Transfer:** This is used for devices such as speaker which needs constant uninterrupted communication between itself and host controller. Typically a fixed bandwidth will be allocated for this communication.

1.3.3.4 Mechanical characteristics

There are several USB connectors' types which are different in size and shape. However, all of them can be divided into 2 series: A and B. There are 2 definitions: Downstream and Upstream. The direction of connection towards host devices is upstream. On the other hand, the direction from host to USB devices is downstream.



Series "A" Connectors

- Series "A" plugs are always oriented **upstream** towards the *Host System*

Series "B" Connectors

- Series "B" plugs are always oriented **downstream** towards the *USB Device*

"A" Plugs (*From the USB Device*)
"A" Receptacles (*Downstream Output from the USB Host or Hub*)
"B" Plugs (*From the Host System*)
"B" Receptacles (*Upstream Input to the USB Device or Hub*)

Fig. 14 Schematics of typical A and B USB connectors [10 chapter 6 page 85]

- "A" receptacle is plugged by "A" plug. "A" receptacle serves as output of host system or USB hub. It is connected downstream towards USB Devices.
- "A" plug is meant to be connected to host "socket" or USB hub downstream port.
- "B" receptacles is plugged by "B" plugs. This serves as upstream input to USB devices or USB hub.
- "B" plug is meant to be connected to USB devices.

For USB 2.0 and 1.0, the cable contains 4 lines as 4 pins on connectors: D+, D-, VBUS, and GND.

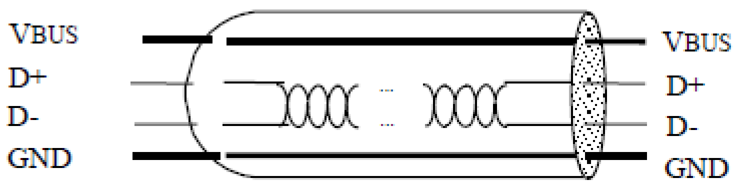


Fig. 15 USB cable [10 chapter 4 page 17]

The VBUS and GND (Voltage and Ground) supply power for the communication, 5V in standard. D+ and D- are data lines. Polarity between two lines is used determine signal logic levels because of its better immunity to electromagnetic noise. Additionally, voltage of Data lines referencing to Ground is used to detect whether the communication is low speed or full speed, and also to detect whether bus is in SE0 state (both lines voltage equals 0). SE0 state is when bus enters Disconnect, Reset or EOP (End-Of-Package).

In USB 3.0, 5 more lines are used for Superspeed transmissions: 1 pair for transmitter, 1 pair for receivers and 1 ground grain. The connector in “A” series is designed to be compatible with USB 1.0 and 2.0. However, in “B” series, only lower versions of connectors can be connected to USB 3.0 receptacle.

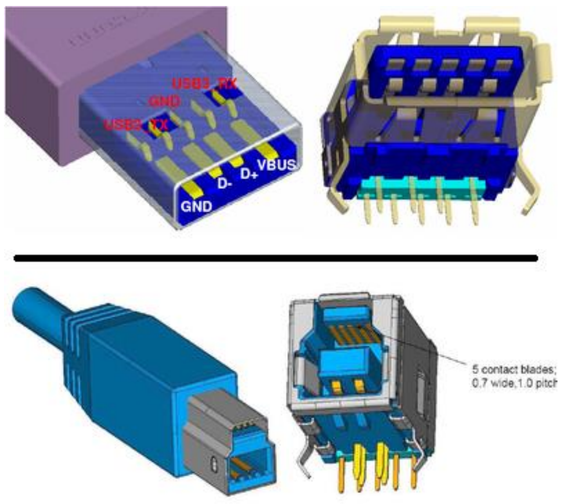


Fig. 16 USB 3.0 “A” plug and receptacle (up) and USB “B” plug and receptacle. Notice the additional pins added. [12]

2 Final Project work: Savonia's Real-time Electrical Measurement

2.1 Overview

This is a big project where Savonia University of Applied Sciences is monitored, stored on database server and presented through Mobile's phone, web, or desktop client. I was in charge of developing the program is written in C#, using .NET framework at the computer collecting data from an ABB M2M box, which measures electricity and related measurements of Savonia, and then sends to database server. The method of sending is implemented using WCF technology. If there is any change in database server, the communication or device’s connection, the settings and configurations can be easily modified in application’s configuration files.

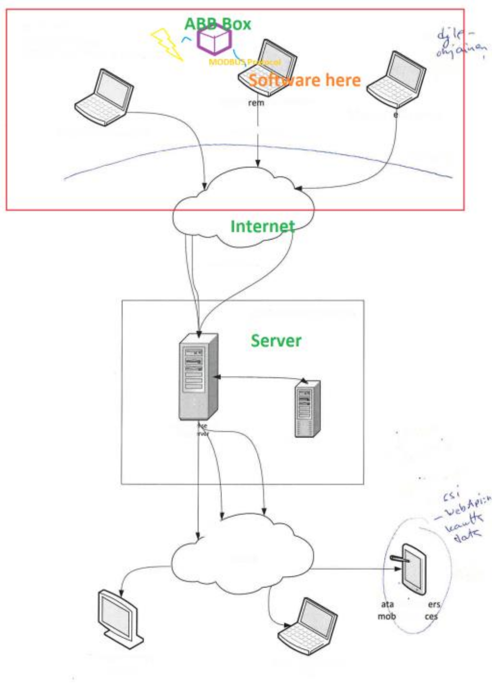


Fig. 17 The architecture of the project including 3 main parts: Data acquisition, Database, Client. My part of the project (in the red box) is the Data Acquisition.

The measuring device in the project ABB M2M using MODBUS protocol over Serial communication. In the next section, I will introduce MODBUS protocol at the level needed for using in the project.

2.2 Introduction to MODBUS Protocol

MODBUS is a messaging application Server/Client protocol published by Modicon (Schneider Electric nowadays) in 1979. Thanks to its simplicity alongside with free cost as well loyalty freedom, it is widely used in the industry. MODBUS protocol can be implemented on TCP over Ethernet media or also over serial communication using RS232 or RS485 standard. In MODBUS serial protocol there are always a Master and several Slave devices. In the easiest explanation, the Master is like a supreme controller which sends query to specific slave to request information as well as configuration commands. If the Slave receives and manages to accomplish the task, information is sent back to the Master to provide the requested data or indicate that the configuration has been done. Format and meaning of this information will be described clear in data-sheet of the devices by manufacturer, and they are usually based roughly on the general design of the MODBUS standard (maybe different from company to company but generally the same).

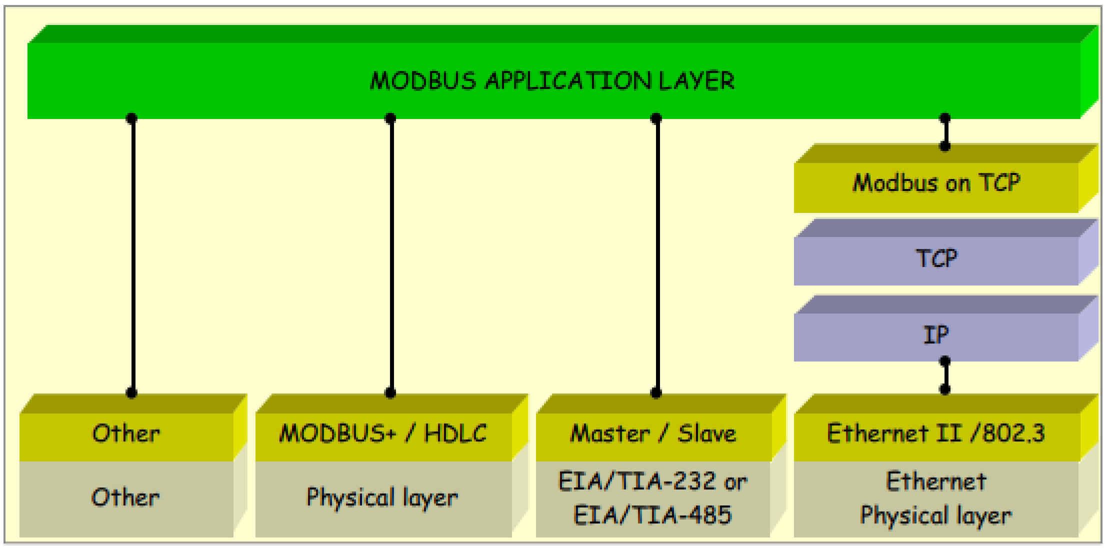


Fig. 18 MODBUS Communication Stack [13 page 2]

All the communications are initiated by the master. From the master, there are two ways it can send message to the slave devices: Unicast and Broadcast.

In unicast mode, the message from master contains the unique address of the device it wants to query from. The device with that unique address sends back a reply.

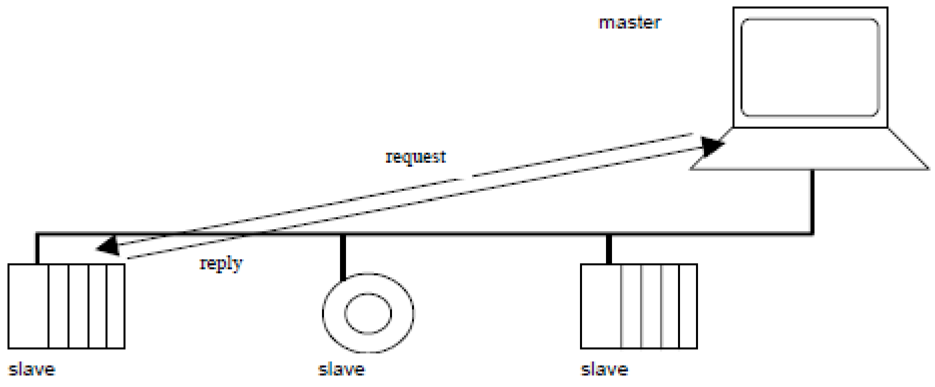


Fig. 19 Unicast mode communication diagram of MODBUS protocol [14 page 7]

In broadcast mode, master sends to all devices on network message, and they will accept it all. No reply from slave device is needed. Address 0 is reserved for this kind of communication.

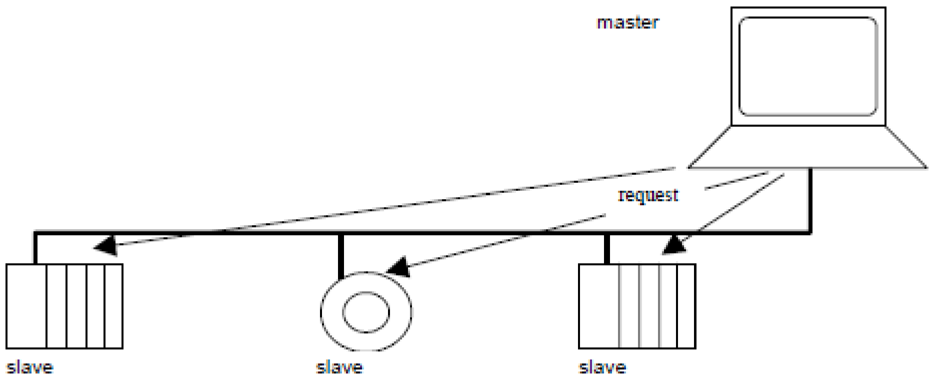


Fig. 20 Broadcast mode communication diagram of MODBUS protocol [14 page 7]

Besides its advantages, MODBUS protocol also has its limitations.

- No super standard to describe or define register address as well as data value meanings. Each manufacturer defines their own rules.
- The maximum devices can be connected to a master is solely 247 [14 page 7].
- Since it is Master/Slave protocol, there is no such way to report changes (Like Event mechanism in several High-Level programming language) and Exception handling. This leads to the Master may have to request over the communication line all the time to monitor the changes if needed. This process may cost the bandwidth heavily.
- The protocol itself is designed quite long time ago, in the 1970s, and introduced in the very late of that era, long binary number such 64 bit integer are not supported.

2.3 RTU MODBUS Communication

As mentioned above, MODBUS can be implemented in multiple media: TCP, Serial Port or MODBUS PLUS high speed token passing network. At this moment we will focus solely on MODBUS communication over serial line, as the electric box supports this method (even though it also supports MODBUS over TCP).

2.3.1 MODBUS RTU & ASCII

There are two serial MODBUS transmissions mode: RTU (Remote Terminal Unit) and ASCII (American Standard Code for Information Interchange). Choosing between these two modes determines how message transmitted over network is packed and decoded.

	MODBUS ASCII MODE	MODBUS RTU MODE
Coding System	Hexadecimal (Each hexadecimal number corresponds for each ASCII character of the message)	8-bit binary (Each 8-bit binary number represents 2 Hexadecimal characters)
Byte autopsy	1 start bit 7 data bits	1 start bit 8 data bits

	1 bit for parity or none 1 stop bit (with parity) or 2 stop bits (without parity)	1 bit for parity, or none 1 stop bit (with parity) or 2 stop bits (without parity)
Error checking	LRC	CRC

Table 4. Comparison of MODBUS ASCII and RTU [15 page 6, 7]

In this project application, we use MODBUS RTU for the communication.

2.3.2 RTU Byte Autopsy

As mentioned in previous section, RTU is used for this project. 8-bit bytes are used in message. Each byte represents 2 hexadecimal characters (therefore 4-bit each). Typically, the 11 bit format (including overheads) in RTU mode will be like this.

Start bit	8 data bits								Parity bit(s)	Stop bit(s)
1	1	1	1	1	1	1	1	1	1	1

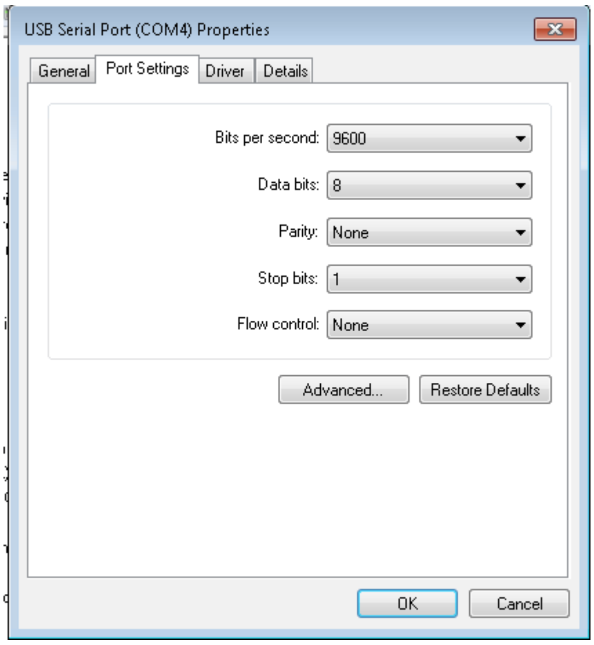
Table 5. Number of bit for each byte of the message in the communication

Stop bits can be set to 1.5 bits or 2 bits.

Parity Bits can be set to Even, Odd or None (the purpose of this is mentioned in Error Checking section 1.2.4 of the report)

The 8 Data bits will be transmitted in LSB first order.

All these settings can be set in the box using manual. Then, the setting of COM port when the device is connected to computer must be set the same correspondingly in Device Manager Settings (Windows).


USB Serial Port (COM4) Properties

General | Port Settings | Driver | Details

Bits per second: 9600
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Fig. 21 COM port setting in Device Manager (Windows)

2.3.3 Message Framing

Each message from master contains multiple bytes, up to 256 max. The frame for a message follows as below.

Slave Device	Function	Data	Error Checking (CRC)

Address	Code		
1 byte	1 byte	N bytes (0 to 252 bytes as the total is 256 max)	2 bytes

Table 6. The frame of a typical message in MODBUS RTU

When transmitting, to let the receiver know when the message is started or ended, the message is placed into Frame with known starting point and ending point which is at least 3.5 character times of silence.

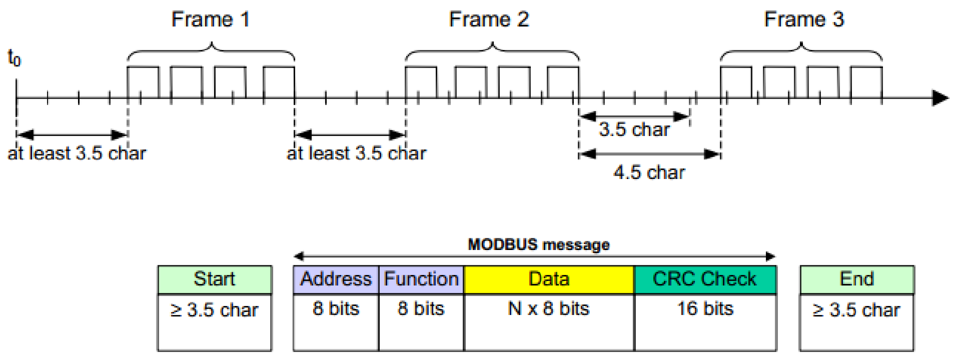


Fig. 22 MODBUS RTU Framing with at least 3.5 character time to separate between 2 messages [14 page 13]

Between 2 bytes in each frame, if the waiting time is more than 1.5 character times, the message will be determined as incomplete and receiver will discard it.

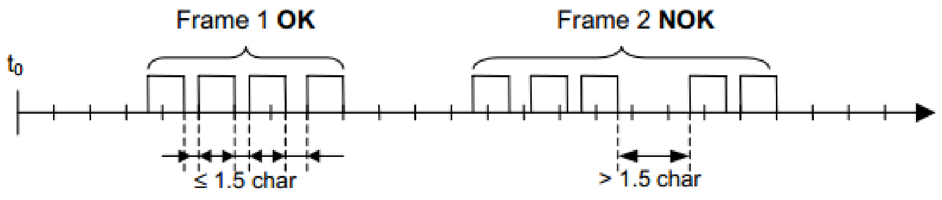


Fig. 23 MODBUS RTU character time determines message is OK or not [14 page 13]

Calculating character time:

For example, with baud rate is 9600, which means 9600 bits per second. Thus, 1 bit costs 1/9600 seconds to be transmitted. Therefore, one byte which is 11 bits costs 11/9600 = 1.14ms.

So message frame value costs 3.5*1.14ms = 4.01ms.

Character frame value costs 1.5*1.14ms = 1.71ms.

Since it is impossible to synchronize 2 devices to exact time frame, the determination of each value is "at least" or "at most", which means the device waits over or less than the time value to determine the message value and its completion.

2.4 Electric Box Set-up

The electrical set-up is not my main part. My physical set-up for the box is solely the part from the ABB box to the computer. ABB M2M supports both RTU and Ethernet protocol. And the mode that we choose for this project is RTU over RS485 lines. At this point, we connects the box straight to the data acquisition computer. Even though, it is supported to connect more devices (up to 247), at this stage, the data will be collected with only one box, which measures the whole Savonia University of Applied Sciences' main power consumption and its related measurements.

In general, the device should be connected as below.

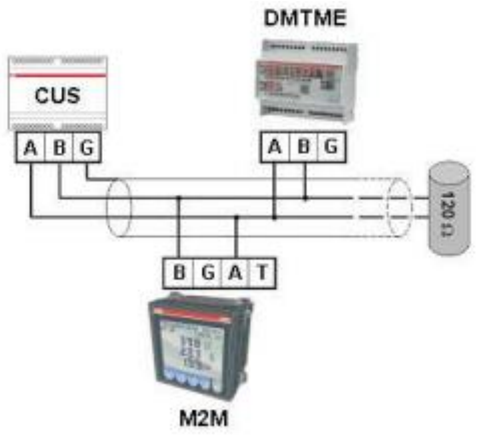


Fig. 24 ABB M2M network connection with a terminal resistor used [16 page 5]

If the line is longer than 500m, a terminal resistor 120 Ohm should be connected to A and B pin.

If the M2M is the end of the connection line, pin T can be connected to pin B to avoid using terminal resistor.

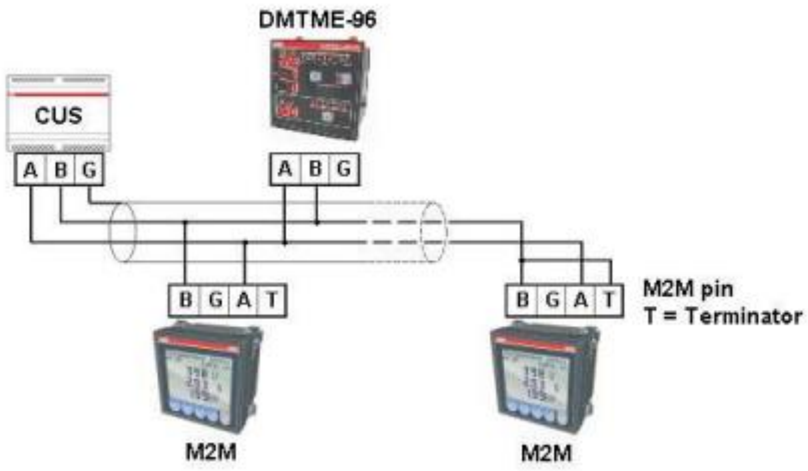


Fig. 25 ABB M2M network connection without terminal resistor [16 page 5]

To connect the computer, modern one without RS232 or RS485 port, we use a RS485 to USB converter made by FTDI: USB-RS485-WE-1800-BT.

Fig. 26 FTDI-USB-RS485-WE-1800-BT [17]

The color wiring of the converter is described as below

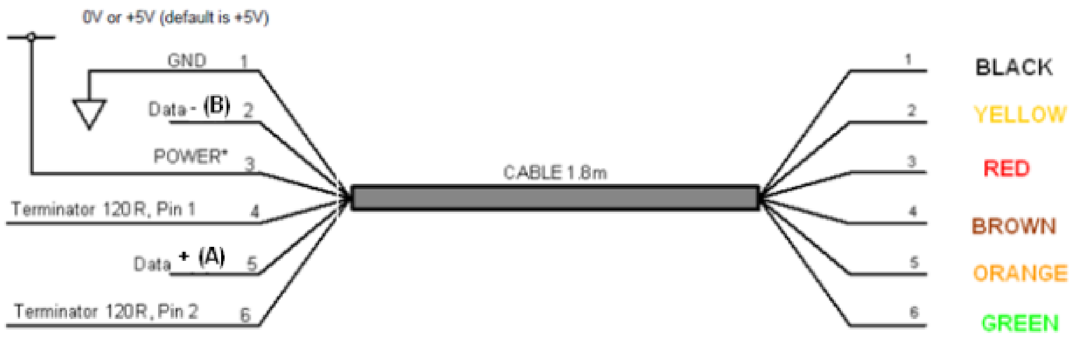


Fig. 27 RS485-to-USB color wiring [18 page 6]

Since we just have one device only in the network. So the Terminator cable will be connected to A and B pin.

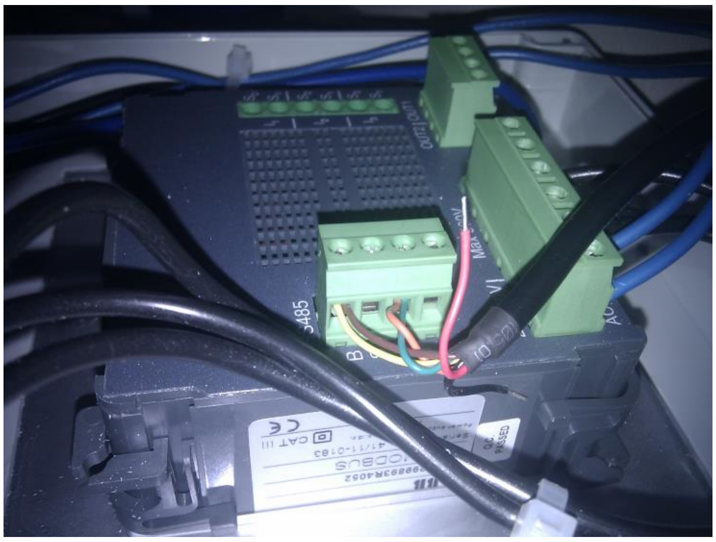


Fig. 28 Wiring of the box and RS485-to-USB converter, A and B pin is used only, Terminator line is connected to them as well

Then the USB cable is connected to the data acquisition PC.

Fig. 29 The ABB M2M box connected to PC

2.5 MODBUS Framework application on ABB M2M

2.5.1 General frame

Like the standard frame discussed in RTU MODBUS Communication section, the frame of the message is below

T1 T2 T3	
Address Field	8 bits
Function code	8 bits
Data field	N*8 bits
Error checker	16 bit CRC
T1 T2 T3	

Table 7. M2M MODBUS frame. T1 T2 T3 represent character times. [16 page 4]

To generate the CRC code, in the M2M MODBUS manual, an algorithm written in C is represented in page 6.

Address Field is the slave unique ID number. This number can be set on the ABB box. It is used to distinguish between devices and is used to query in unicast mode by the master device which is the data acquisition computer in this case.

ABB M2M box supports 3 functions:

- 03h: Reading Holding Register
- 10h: Writing Parameters
- 11h: Report slave ID

In this project, the C# application uses function 03h to query the data from the ABB box.

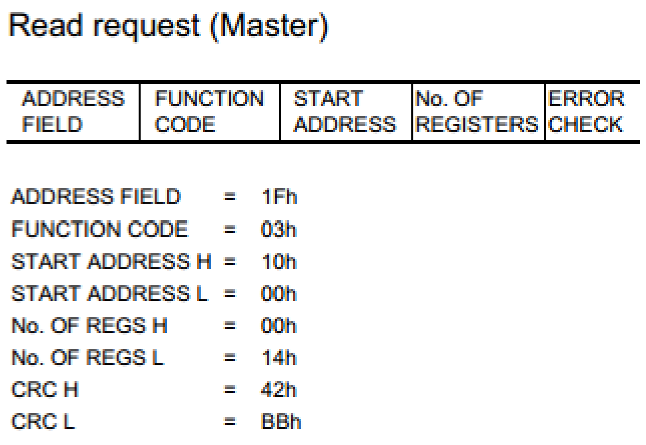
Data field contains the address of start register and number of consecutive ones to read that contains the information the master wants to query.

2.5.2 Function 03h

In MODBUS protocol standard document [15 page 28], function 3h is meant to "Read holding Registers". It is able to read consecutive adjacent registers, which are 2-word in size each. For ABB M2M, it can read max 24 consecutive registers or 24 measurements at a time.

As described in "Introduction to MODBUS Protocol" section, this function is Unicast mode communication. There are two parts of a query: request from master and reply from the slave.

2.5.2.1 Read Request from the Master



Read request (Master)

ADDRESS FIELD	FUNCTION CODE	START ADDRESS	No. OF REGISTERS	ERROR CHECK

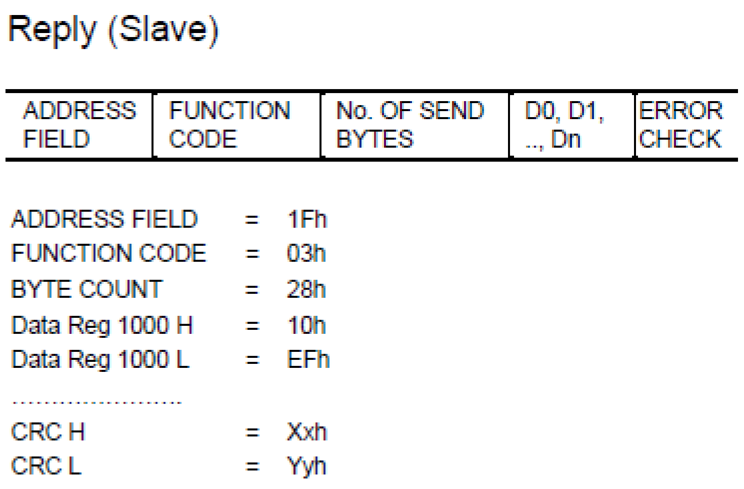
ADDRESS FIELD	=	1Fh
FUNCTION CODE	=	03h
START ADDRESS H	=	10h
START ADDRESS L	=	00h
No. OF REGS H	=	00h
No. OF REGS L	=	14h
CRC H	=	42h
CRC L	=	BBh

Fig. 30 Function 03h Reading Request Message example [16 page 7]

In the example, the ID address of the slave is 1Fh (31d in decimal), the message using function code 03h (3d), starting address of the register is 1000h (4096d), read 14 consecutive registers, and the CRC is 42BBh.

2.5.2.2 Reply from slave device

The corresponding slave device will reply in this format.



Reply (Slave)

ADDRESS FIELD	FUNCTION CODE	No. OF SEND BYTES	D0, D1, .., Dn	ERROR CHECK

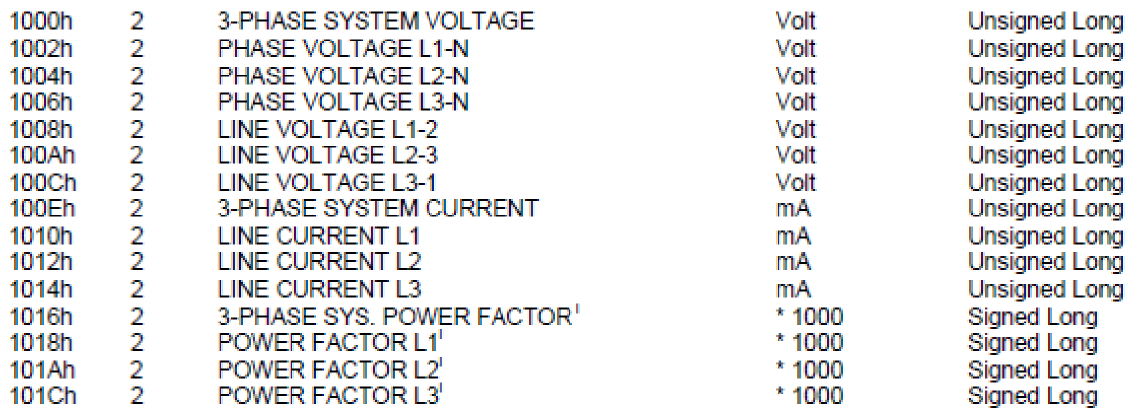
ADDRESS FIELD	=	1Fh
FUNCTION CODE	=	03h
BYTE COUNT	=	28h
Data Reg 1000 H	=	10h
Data Reg 1000 L	=	EFh
....		
CRC H	=	Xxh
CRC L	=	Yyh

Fig. 31 Function 03h Reply Message example [16 page 7]

In the example, the address shows the responding device ID, the function code, number of data bytes count (2xNumber of register). Then number of data byte fields' value corresponding to the number of registers requested.

2.5.3 Register Meaning

On page 9 and 10 ABB M2M MODBUS RTU manual, information to check what the available register are and its corresponding meaning is listed. From this list, we choose the essential measurement to be read and stored in the database server and put them in the configuration file of the C# application to poll.



1000h	2	3-PHASE SYSTEM VOLTAGE	Volt	Unsigned Long
1002h	2	PHASE VOLTAGE L1-N	Volt	Unsigned Long
1004h	2	PHASE VOLTAGE L2-N	Volt	Unsigned Long
1006h	2	PHASE VOLTAGE L3-N	Volt	Unsigned Long
1008h	2	LINE VOLTAGE L1-2	Volt	Unsigned Long
100Ah	2	LINE VOLTAGE L2-3	Volt	Unsigned Long
100Ch	2	LINE VOLTAGE L3-1	Volt	Unsigned Long
100Eh	2	3-PHASE SYSTEM CURRENT	mA	Unsigned Long
1010h	2	LINE CURRENT L1	mA	Unsigned Long
1012h	2	LINE CURRENT L2	mA	Unsigned Long
1014h	2	LINE CURRENT L3	mA	Unsigned Long
1016h	2	3-PHASE SYS. POWER FACTOR[1]	* 1000	Signed Long
1018h	2	POWER FACTOR L1[1]	* 1000	Signed Long
101Ah	2	POWER FACTOR L2[1]	* 1000	Signed Long
101Ch	2	POWER FACTOR L3[1]	* 1000	Signed Long

Fig.32 Registers meaning, type of number, theirs registers asserted from the manual [16 page 9]

Take the first reading from the list as an example, this reading shows the 3-phase voltage of the system, in Volt. The address of it is 1000h (4096d), 2-word in size, and this number is unsigned long (32-bit).

All the measurements are 32-bit long, but are either Unsigned or Signed. If it is signed, it will be expressed in Two Complement Format.

All the addresses is converted into decimal number for easier to read in the program. Thus, the settings in configuration are like below.

Address (in decimal)	Number of word	Meaning and Unit	Type (signed or unsigned)
4096	2	3-PHASE SYSTEM VOLTAGE Volt	unsigned
4098	2	PHASE VOLTAGE L1-N Volt	unsigned
4100	2	PHASE VOLTAGE L2-N Volt	unsigned
4102	2	PHASE VOLTAGE L3-N Volt	unsigned

Table 8. Example of reading configuration with addresses converted into Decimal

2.6 .NET Framework C# application

2.6.1 Introduction to .NET framework

.NET framework is provided by Microsoft to be the platform for develop Windows related software application. It contains libraries of interfaces, classes, connectivity, network services... for purpose of making it easy for developer.

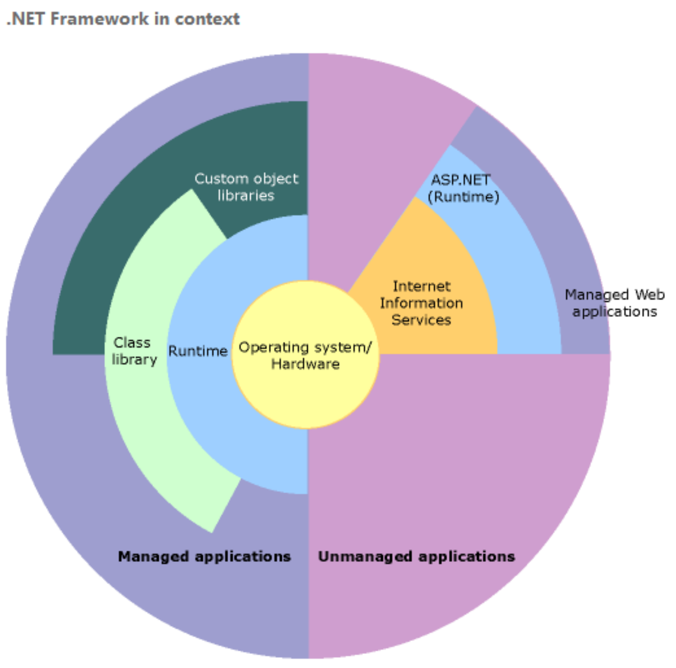


Fig. 33 Main features of .NET framework [19]

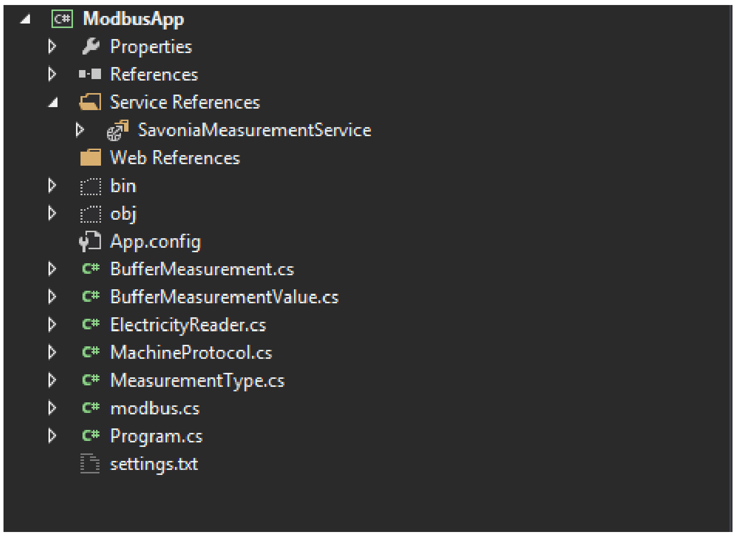
In this project, my part software on the data acquisition of computer is developed in .NET framework API level 4.5, written in C#, using Visual Studio 2013 IDE (also provided by Microsoft).

It is a simple console application polling data from electrical ABB box. Therefore, user interface in minimal. The main aspect of .NET framework used in the project discussed for this report is “SerialPort” Class (in System.IO.Ports) to read data from serial communication.

SerialPort class helps access to COM-port device from Windows without programming in low level. The library provides properties, methods and events to establish, read, write and close communication over serial port. This class was added since API level 2.0 of .NET framework.

2.6.2 The Application Structure

The software is a console application which reads the data from ABB box continuously at a predefined interval after started. Then user only need to press any key to stop it. As mentioned before, the user interface is minimized for the ease of use and maximum of performance. The structure of the software is also optimized so that it is easy to maintain and modified if needed. The hierarchy of the console project named ModbusApp is as below.



ModbusApp
Properties
References
Service References
SavoniaMeasurementService
Web References
bin
obj
App.config
BufferMeasurement.cs
BufferMeasurementValue.cs
ElectricityReader.cs
MachineProtocol.cs
MeasurementType.cs
modbus.cs
Program.cs
settings.txt

Fig. 34 Hierarchy of ModbusApp project

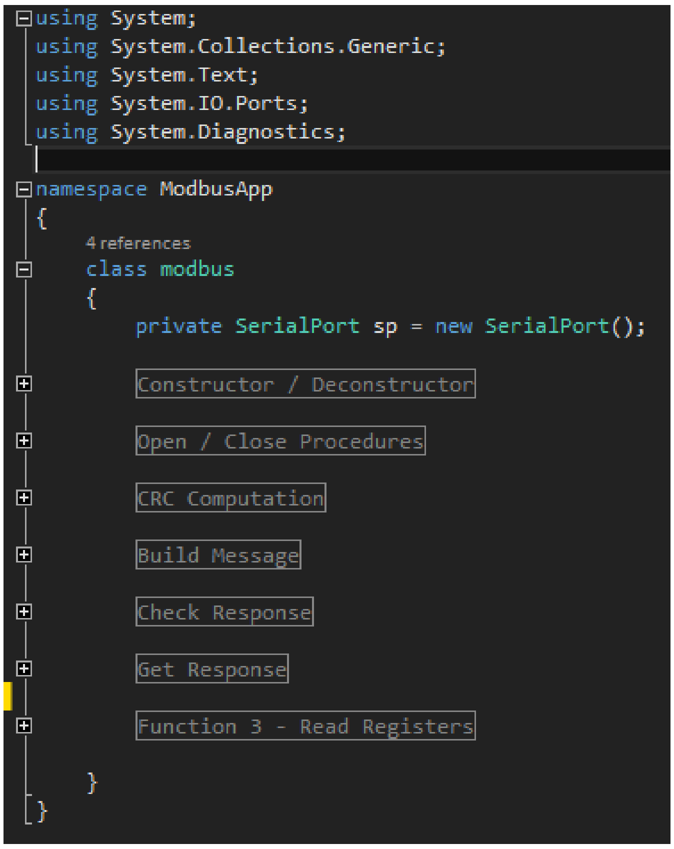
Component	Description
“Modbus” Class	The core component of the programme: providing all necessary utilities to access device through Modbus protocol using System.IO.Ports of .NET library
“MachineProtocol” Class	This class contains all properties for Serial Port connection such Port name, Baud rate, stop bits, Sample Rate, ABB Box ID, Read function code, Measurement ID and Provider ID & Passcode on Database server.
“MeasurementType” Class	This class contains address, number of register, format of number (signed or unsigned) of each wanted measurement register.
“BufferMeasurement” Class	This class contains measuring date, machine ID number, and list of measurement (BufferMeasurementValue). This class is used to serialize to save to local drive in case of the program having problem of sending data to server.
“BufferMeasurementValue” Class	This class contains the pair of Register Address & value, meaning the actual measurement value. This is used for each element of MeasurementList in “BufferMeasurement” which is used for serialization purpose.
“ElectricityReader” Class	The ultimate main class of interacting with the device. It contains all the needed properties, methods helping accessing to the device, starting and stopping the communication. Users just have to call 2 methods the

	use the class: `StartReader`() and `StopReader`() to start and stop reading accordingly. All the settings will be changed in **App.config** file (all of these configurations are read into "MachineProtocol" class) in xml format and **settings.txt** file (most of these settings are read into "`MeasurementType`" class) in text format. Syntax of these files will be explained clearer later.
App.config	Contains configurations of the program and WCF configurations for communication to database server.
Settings.txt	Contains ABB box ID, read function code, and list of reading register address, number of words, meaning, number type, and corresponding sensor ID of it on database server.
Program.cs	This is main logic structure of the application. It is minimized to the simplest. Creating an `ElectricityReader` instance, Loading all settings, start reader, and waiting for any key pressed to stop.
SavoniaMeasurementService	This is the reference to the classes created by another person who are in charge of database server. These classes contains properties for storing data and methods needed for sending to database server using WCF technology.

Table 9. Main components ModbusApp project

2.6.3 "Modbus" Class

This class is modified from a class provided inside a CodeProject tutorial of how to program MODBUS communication in C# using .NET framework at API level 2.0. The author username is "distantcity" and the link of the tutorial is http://www.codeproject.com/Articles/20929/Simple-Modbus-Protocol-in-C-NET-2-0



```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO.Ports;
using System.Diagnostics;

namespace ModbusApp
{
    4 references
    class modbus
    {
        private SerialPort sp = new SerialPort();

        Constructor / Deconstructor

        Open / Close Procedures

        CRC Computation

        Build Message

        Check Response

        Get Response

        Function 3 - Read Registers

    }
}
```

Fig. 35 Modbus Class Hierarchy

The class contains on property called sp, instancing from “SerialPort” Class. This is the basic component to access to COM-Port.

Methods:

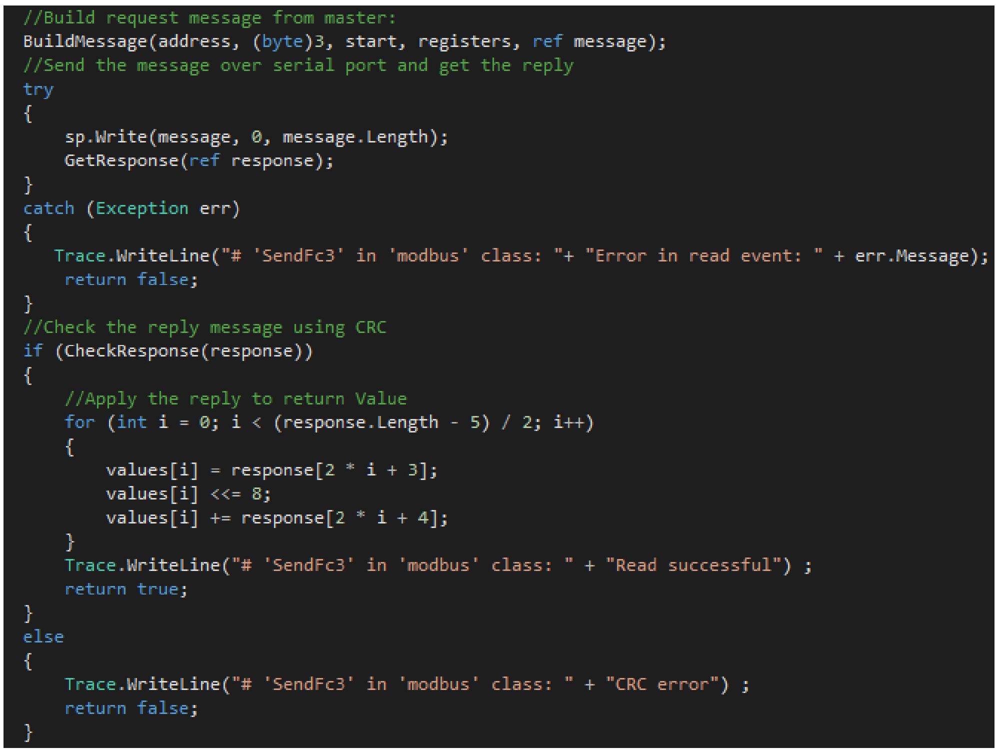
Region	Methods	Description
Constructor / Descontructor	`modbus()`	Default Constructor Method
	`~modbus()`	Default Desctructor Method
Open / Close Procedures	`public bool Open(string portName, int baudRate, int databits, Parity parity, StopBits stopBits)`	Open port with the setting in parameters. • `portName` typically is COMxx (we will use COM4) • `baudrate` typically is 9600 • `databits` typically is 8 • `parity` typically is none • `stopbits` typically is none These settings must be the same as what are set in Device Manager and on the box.
	`public bool Close()`	Close SerialPort Component
CRC Computation	`private void GetCRC(byte[] message, ref byte[] CRC)`	This method calculates the CRC (Cyclic Redundancy Check) values from the message and stores it in CRC byte array.
Build Message	`private void BuildMessage(byte address, byte type, ushort start, ushort registers, ref byte[] message)`	This method builds the exact request message that is sent to the slave device through serial port. This will be called in SendFc3(which means send

		function code 3) • `address` is the address of the device, or MachineID • `type` is function type such 3 is Read Register • `start` is the starting register for reading • `registers` is the number of adjacent registers to be read • `message` is a byte array which is sent over the serial port
Check Response	`private bool CheckResponse(byte[] response)`	This method checks the response by calculating the CRC values from it and compares the calculated values with the CRC component at the end of the response.
Get Response	`private void GetResponse(ref byte[] response)`	This methods is called in the SendFc3 method, right after the request message is sent over serial port. This read bytes from the reply from the corresponding slave device.
Function 3- Read Registers	`public bool SendFc3(byte address, ushort start, ushort registers, ref ushort[] values)`	• This method checks the sp (SerialPort) component opened or not. Then the method builds the message with information provided in the parameters and sends it over the serial line. • Then it reads the response on serial port, checks error with CRC calculation and stores the result in "`values`" byte array

Table 10 Methods of "Modbus" class

Note: the "`values`" array is in ushort since then will be shifted afterward in integer number to create Unsigned 32-bit long or Signed 32-bit long number. If using short type only, this may result problem of unexpected negative number. This is one problem that we encountered during the project.

2.6.3.1 Inside SendFc3 method



```
//Build request message from master:
BuildMessage(address, (byte)3, start, registers, ref message);
//Send the message over serial port and get the reply
try
{
    sp.Write(message, 0, message.Length);
    GetResponse(ref response);
}
catch (Exception err)
{
   Trace.WriteLine("# 'SendFc3' in 'modbus' class: "+ "Error in read event: " + err.Message);
    return false;
}
//Check the reply message using CRC
if (CheckResponse(response))
{
    //Apply the reply to return Value
    for (int i = 0; i < (response.Length - 5) / 2; i++)
    {
        values[i] = response[2 * i + 3];
        values[i] <<= 8;
        values[i] += response[2 * i + 4];
    }
    Trace.WriteLine("# 'SendFc3' in 'modbus' class: " + "Read successful") ;
    return true;
}
else
{
    Trace.WriteLine("# 'SendFc3' in 'modbus' class: " + "CRC error") ;
    return false;
}
```

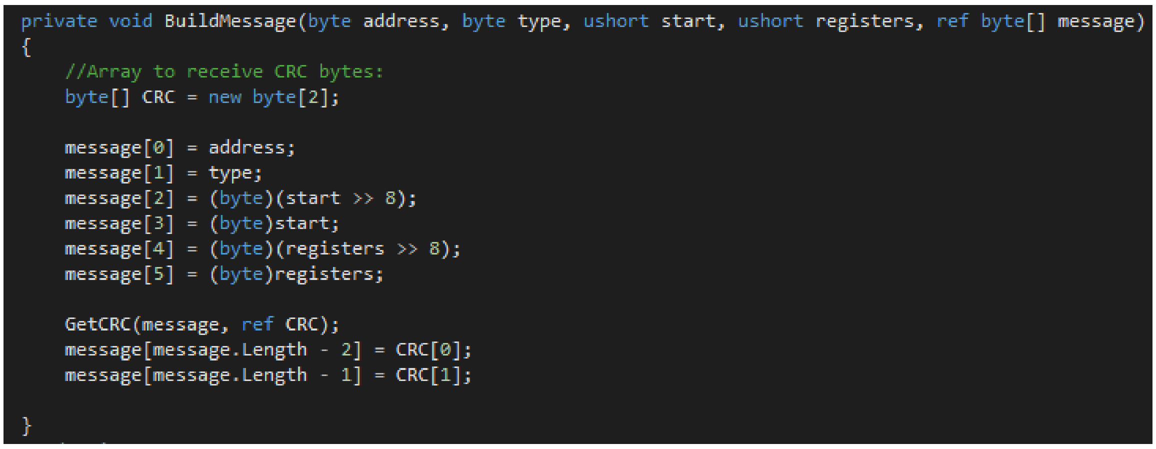
Fig. 36 Part of SendFc3 method

The message to send over serial line is made through `BuildMessage` method with "`address`", "`start`" (start register address) and "`registers`" (number of adjacent registers to read) parameters. The message value is applied to "`message`" byte array (8-byte in length).

The message is sent using Write method in SerialPort class. The reply is read and pushed into "`response`" byte array.

Then the reply is checked with CRC. If it is ok, the result will be applied to "`values`".

2.6.3.2 Inside BuildMessage method



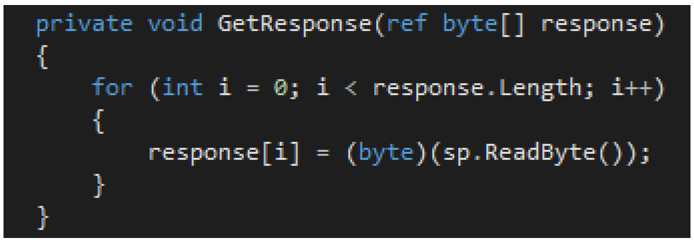
```
private void BuildMessage(byte address, byte type, ushort start, ushort registers, ref byte[] message)
{
    //Array to receive CRC bytes:
    byte[] CRC = new byte[2];

    message[0] = address;
    message[1] = type;
    message[2] = (byte)(start >> 8);
    message[3] = (byte)start;
    message[4] = (byte)(registers >> 8);
    message[5] = (byte)registers;

    GetCRC(message, ref CRC);
    message[message.Length - 2] = CRC[0];
    message[message.Length - 1] = CRC[1];

}
```

Fig. 37 BuildMessage method

The address, function code, start register, number of adjacent registers to read are applied to each byte of "message" array.

CRC values are calculated and applied to 2 last bytes.

2.6.3.3 Inside GetResponse method

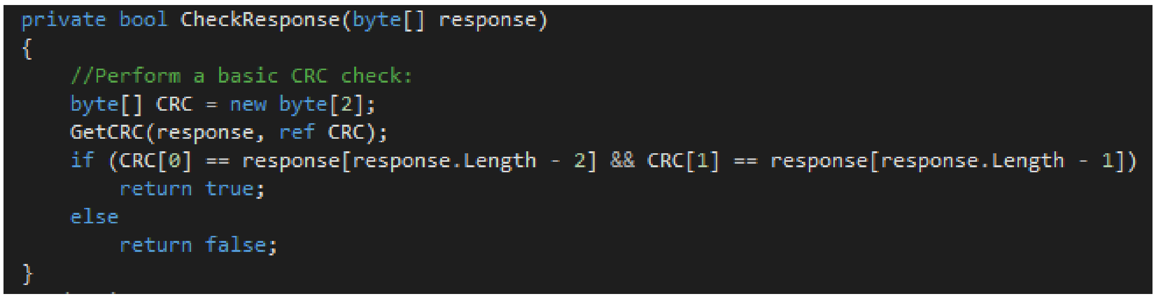


```
private void GetResponse(ref byte[] response)
{
    for (int i = 0; i < response.Length; i++)
    {
        response[i] = (byte)(sp.ReadByte());
    }
}
```

Fig. 38 GetResponse Method

Each byte of “response” byte array is read using ReadByte method in SerialPort class.

2.6.3.4 Inside CheckResponse method



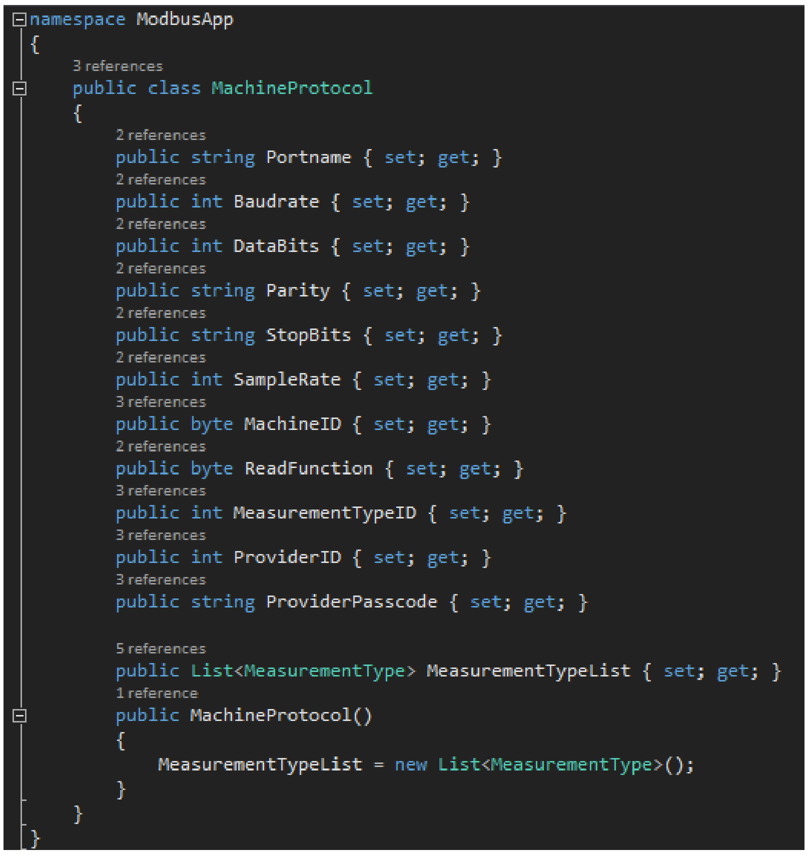
```
private bool CheckResponse(byte[] response)
{
    //Perform a basic CRC check:
    byte[] CRC = new byte[2];
    GetCRC(response, ref CRC);
    if (CRC[0] == response[response.Length - 2] && CRC[1] == response[response.Length - 1])
        return true;
    else
        return false;
}
```

Fig. 39 CheckResponse method

The CRC values are calculated with the response byte array. Then the result is compared with the CRC part of the response (last two bytes). If they are matched, the method returns true. Otherwise, the method returns false.

2.6.4 “MachineProtocol” Class

This class basically loaded all the configurations and settings of the application when started. It contains only properties and is instanced and used in “ElectricityReader” class.



```
namespace ModbusApp
{
    3 references
    public class MachineProtocol
    {
        2 references
        public string Portname { set; get; }
        2 references
        public int Baudrate { set; get; }
        2 references
        public int DataBits { set; get; }
        2 references
        public string Parity { set; get; }
        2 references
        public string StopBits { set; get; }
        2 references
        public int SampleRate { set; get; }
        3 references
        public byte MachineID { set; get; }
        2 references
        public byte ReadFunction { set; get; }
        3 references
        public int MeasurementTypeID { set; get; }
        3 references
        public int ProviderID { set; get; }
        3 references
        public string ProviderPasscode { set; get; }

        5 references
        public List<MeasurementType> MeasurementTypeList { set; get; }
        1 reference
        public MachineProtocol()
        {
            MeasurementTypeList = new List<MeasurementType>();
        }
    }
}
```

Fig. 40 “MachineProtocol” Class hierarchy

The meaning of each property is self-defined by its name. Each of these property is read from either App.config or settings.txt. App.config is for unique single configuration while settings is for generic one such as list of measurements' addresses and its related fields.

Property	Description	Location to read from
`Portname`	Typically COMxx (in this project, this is default COM4) This values can be set in Device Manager in settings of COM port)	App.config
`Baudrate`	Default 9600 (also can be set in Device Manager)	
`Databits`	Typically 8 (also can be set in Device Manager)	
`Parity`	Typical none (also can be set in Device Manager)	
`StopBits`	Typical none (also can be set in Device Manager)	
`SampleRate`	In milliseconds (3000 = 3 seconds)	
`MeasurementTypeID`	These three values are used for authentication to send measurement values to database server.	
`ProviderID`		
`ProviderPasscode`		
`MachineID`	This is the Slave ID of the ABB M2M box (31	Settings.txt

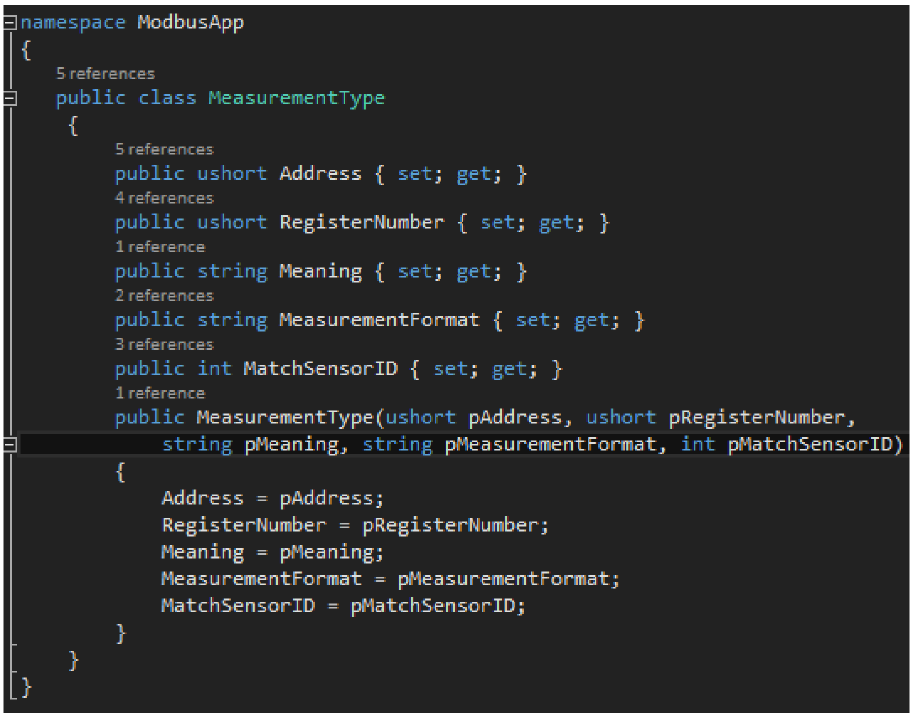
	is used in this project) (this setting must be set correspondingly with the ID setting on the ABB M2M box).	
`ReadFucntion`	This is 3 here for reading. To use SendFc3 in Modbus (besides 3 there are several more reading function, but for this project, function code 3 is enough).	
`MeasurementTypeList`	List of MeasurementType, each of this type includes Register address, number of register, and format of it (unsigned or signed number) and corresponding sensor ID on database storing class.	

Table 11. Properties of "MachineProtocol" Class

2.6.5 "MeasurementType" Class

This class stores each measurement's information such as address, number of adjacent registers, meaning, format (signed or unsigned) and corresponding Sensor ID on database server's storing class.

This is used for each element of the `MeasurementTypeList` generic list inside "`MachineProtocol`" Class.



```
namespace ModbusApp
{
    5 references
    public class MeasurementType
     {
        5 references
        public ushort Address { set; get; }
        4 references
        public ushort RegisterNumber { set; get; }
        1 reference
        public string Meaning { set; get; }
        2 references
        public string MeasurementFormat { set; get; }
        3 references
        public int MatchSensorID { set; get; }
        1 reference
        public MeasurementType(ushort pAddress, ushort pRegisterNumber,
            string pMeaning, string pMeasurementFormat, int pMatchSensorID)
        {
            Address = pAddress;
            RegisterNumber = pRegisterNumber;
            Meaning = pMeaning;
            MeasurementFormat = pMeasurementFormat;
            MatchSensorID = pMatchSensorID;
        }
    }
}
```

Fig. 41 "MeasurementType" Class Hierarchy

Properties	Description
`Address`	Start address of the first register to be read
`RegisterNumber`	Number of adjacent register to be read

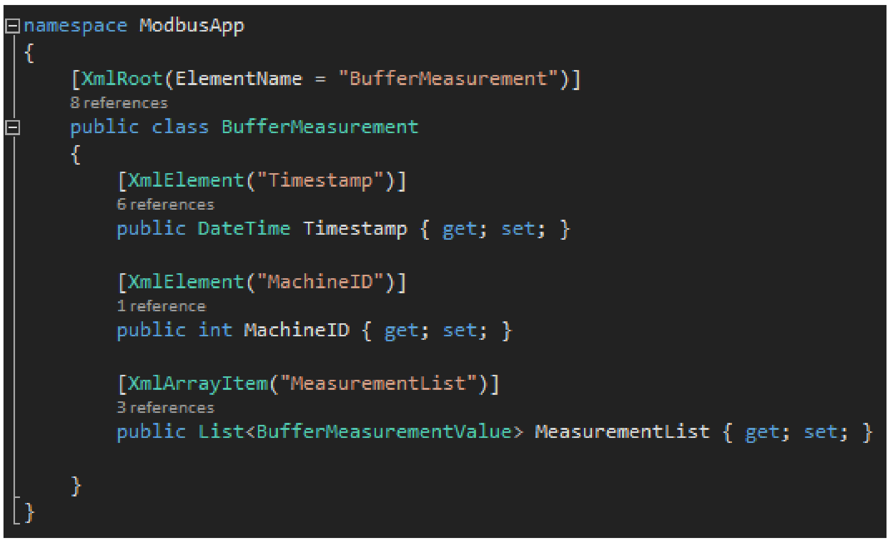
`Meaning`	The meaning of the reading value
`MeasurementFormat`	Format of the number: signed long or unsigned long
`MatchSensorID`	The corresponding sensor ID in the storing class on Database Server

Table 12. Properties of "MeasurementType" Class

Besides, there is one customized constructor with parameters: `public MeasurementType(ushort pAddress, ushort pRegisterNumber, string pMeaning, string pMeasurementFormat, int pMatchSensorID)`

2.6.6 "BufferMeasurement" Class

This class and `BufferMeasurementValue` is used for serialization purpose in case of problem in connection resulting measurements cannot be sent to the database server. In that scenario, the measurements will be saved on local drive, at a location defined in App.config. Then when the next sending is successful, the application will send all the measurement inside saved location (or "buffer" folder).



```
namespace ModbusApp
{
    [XmlRoot(ElementName = "BufferMeasurement")]
    8 references
    public class BufferMeasurement
    {
        [XmlElement("Timestamp")]
        6 references
        public DateTime Timestamp { get; set; }

        [XmlElement("MachineID")]
        1 reference
        public int MachineID { get; set; }

        [XmlArrayItem("MeasurementList")]
        3 references
        public List<BufferMeasurementValue> MeasurementList { get; set; }

    }
}
```

Fig. 42 "BufferMeasurement" Class hierarchy

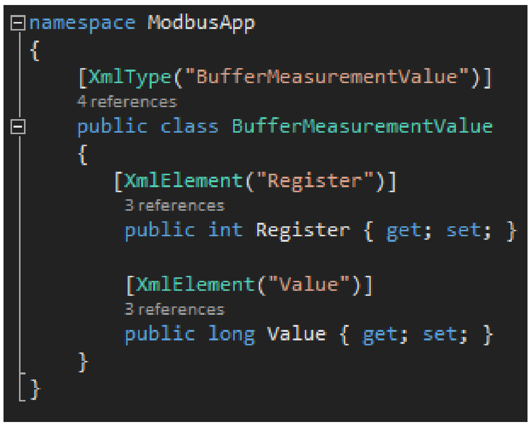
Note: above each class definition and each property, there is tag definition required for serialization.

Properties	Description
`Timestamp`	Data and time of measurements (this will be saved in GMT)
`MachineID`	ID of the slave device
`MeasurementList`	List of measuremnts (each element is a "BufferMeasurementValue")

Table 13. Properties of "BufferMeasurement" class

2.6.7 "BufferMeasurementValue" Class

This class contains 2 properties: `Register` and `Value`, which is the address of measurement start register and its value.



```
namespace ModbusApp
{
    [XmlType("BufferMeasurementValue")]
    4 references
    public class BufferMeasurementValue
    {
        [XmlElement("Register")]
        3 references
        public int Register { get; set; }

        [XmlElement("Value")]
        3 references
        public long Value { get; set; }
    }
}
```

Fig. 43 “BufferMeasurementValue” Class hierarchy

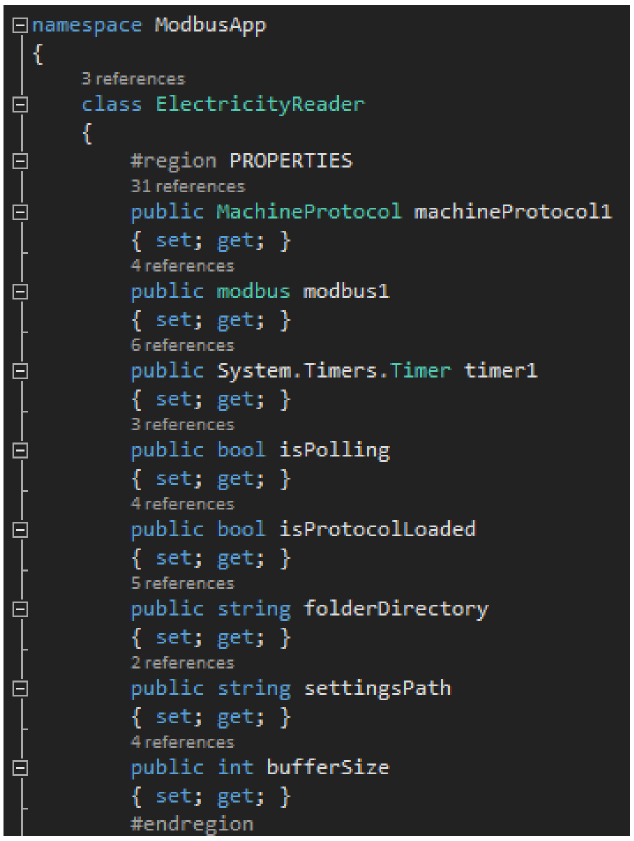
Properties	Description
`Register`	The address of measurement, its meaning is stated in datasheet of the ABB M2M (this is in decimal, converted from Hexadecimal number in manual)
`Value`	The value of measurement

Table 14. Properties of “BufferMeasurementValue” Class

2.6.8 “ElectricityReader” Class

This is the most important class of the program. It serves as the back bone of it. In the main program, an instance of “ElectricityReader” is needed. The programmer just has to call `StartReader` and `StopReader` method to start and stop requesting and reading measurement over serial port. In the application, `StartReader` is called when the program is started. And `StopReader` is called when user press any key to stop the application.

Inside the class, multiple properties and methods are provided for all functionality such as accessing serial port, loading configurations, sending data to database server and serializing data to local drive when needed.

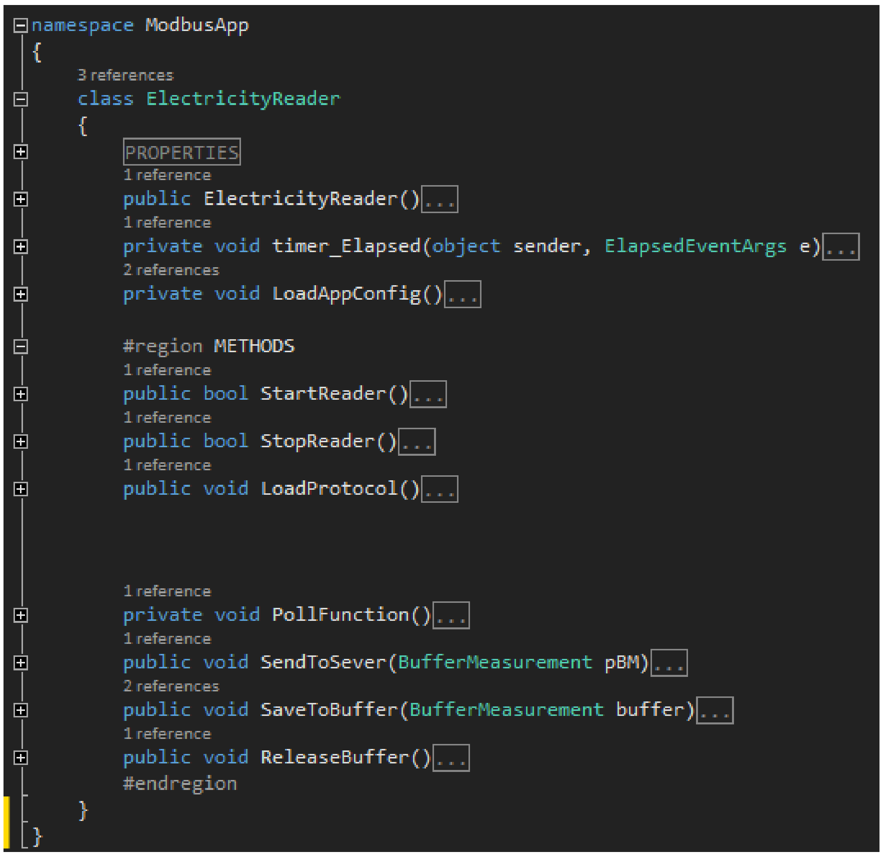


```
namespace ModbusApp
{
    3 references
    class ElectricityReader
    {
        #region PROPERTIES
        31 references
        public MachineProtocol machineProtocol1
        { set; get; }
        4 references
        public modbus modbus1
        { set; get; }
        6 references
        public System.Timers.Timer timer1
        { set; get; }
        3 references
        public bool isPolling
        { set; get; }
        4 references
        public bool isProtocolLoaded
        { set; get; }
        5 references
        public string folderDirectory
        { set; get; }
        2 references
        public string settingsPath
        { set; get; }
        4 references
        public int bufferSize
        { set; get; }
        #endregion
```

Fig. 44 Property Hierarchy of “ElectricityReader” class

Property	Description
`machineProtocol1`	This is the instance of “MachineProtocol” Class, storing all configurations and settings of program from App.config and settings.txt when program loaded.
`modbus1`	This is the instance of “Modbus” class, which is in charge of communicating to the ABB M2M box over serial line.
`timer1`	Timer for polling, the sample rate is set in through this object
`isPolling`	Boolean variable showing the program is reading or not
`isProtocolLoaded`	Boolean variable showing the program load protocol or not
`folderDirectory`	The folder directory where the buffer will be stored. (this is defined in App.config)
`settingPath`	The path of the settings.txt file (this is defined in App.config)
`bufferSize`	The number of measurements in buffer

Table 15. Properties of “ElectricityReader” Class



```
namespace ModbusApp
{
    3 references
    class ElectricityReader
    {
        PROPERTIES
        1 reference
        public ElectricityReader()...
        1 reference
        private void timer_Elapsed(object sender, ElapsedEventArgs e)...
        2 references
        private void LoadAppConfig()...

        #region METHODS
        1 reference
        public bool StartReader()...
        1 reference
        public bool StopReader()...
        1 reference
        public void LoadProtocol()...



        1 reference
        private void PollFunction()...
        1 reference
        public void SendToSever(BufferMeasurement pBM)...
        2 references
        public void SaveToBuffer(BufferMeasurement buffer)...
        1 reference
        public void ReleaseBuffer()...
        #endregion
    }
}
```

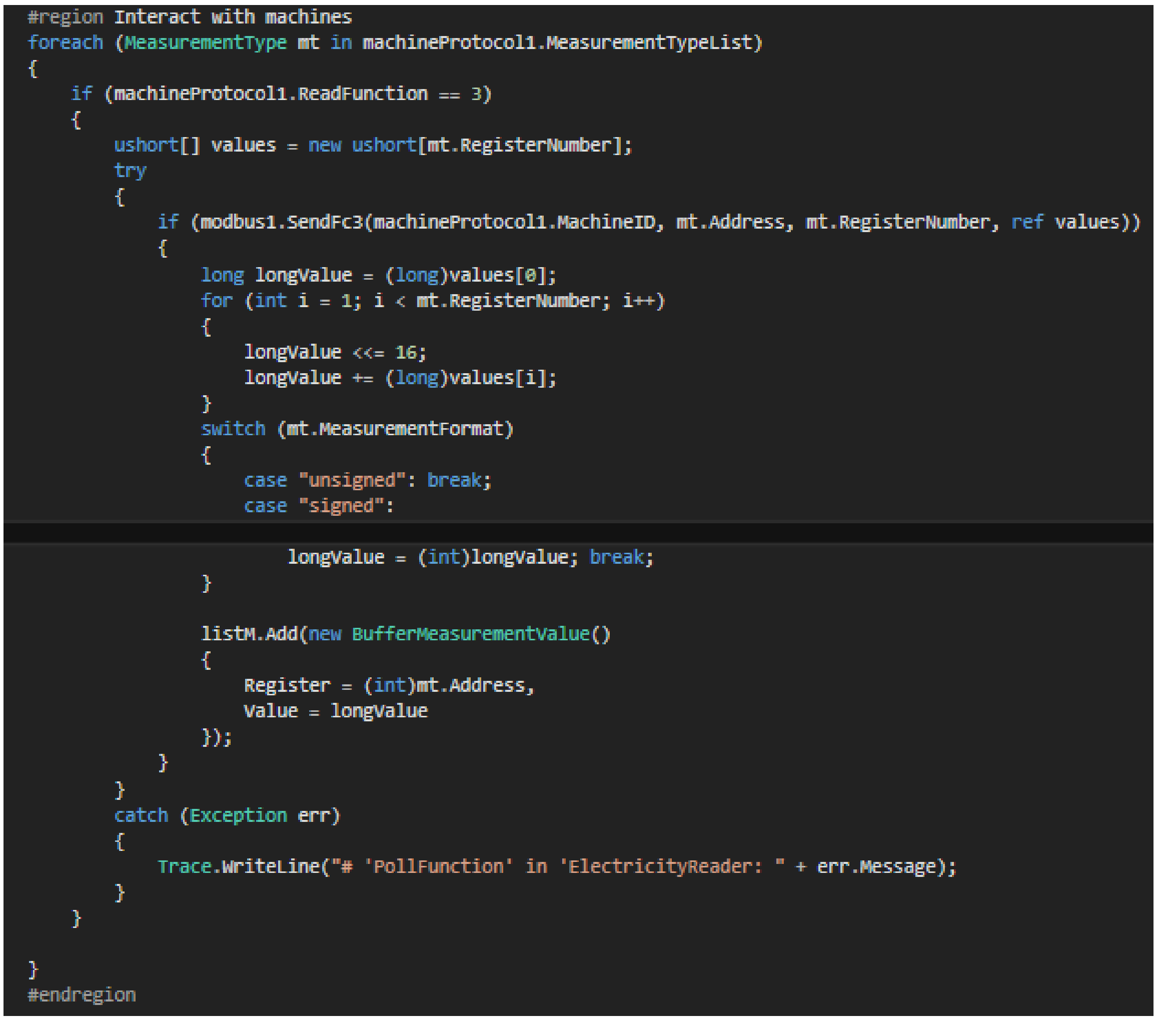
Fig. 45 Method Hierarchy of “ElectricityReader” Class

Method	Description
`ElectricityReader()`	Constructor which initializes all properties
`LoadAppConfig`	This method is used to load configurations from App.config, to machineprotocol1 property
`LoadProtocol`	This method is used to load generic settings in settings.txt
`timer_Elapsed`	This method is attached to timer1's Elapsed Event Handler. Thus, it is fired at every interval time defined in timer1. Inside the method, it simply calls `PollFunction`, which is in charge of sending request message and reading reply message from ABB M2M box.
`StartReader`	Start to poll measurement from the box
`StopReader`	Stop polling measurements from the box
`PollFucntion`	When starting the reader, the timer will start. This will be called when the timer is elapsed. This method is used to send request message to slave device and read reply from it. It puts Time stamping to measurement and sends it to server using “`SendToServer`” method. If the measurement cannot be sent, the data will be stored to the buffer on local drive using “`SaveToBuffer`” method.
`SendToServer`	Send the measurement to server, using WCF technology. The

	storing data class on data server is referenced in "`SavoniaMeasurementService`" component
`SaveToBuffer`	If there are some problems resulting the measurements cannot be sent to server, the measurements will be serialized, saved to local drive. The program continue to do so as long as the problem is still there. When the next measurement is sent successfully, the buffer will be released to the server using "`ReleaseBuffer`" method.
`ReleaseBuffer`	This method de-serializes all the measurements saved in buffer folder and sends them to server

Table 16. Methods of "ElectricityReader" class

Inside PollFunction method



```
#region Interact with machines
foreach (MeasurementType mt in machineProtocol1.MeasurementTypeList)
{
    if (machineProtocol1.ReadFunction == 3)
    {
        ushort[] values = new ushort[mt.RegisterNumber];
        try
        {
            if (modbus1.SendFc3(machineProtocol1.MachineID, mt.Address, mt.RegisterNumber, ref values))
            {
                long longValue = (long)values[0];
                for (int i = 1; i < mt.RegisterNumber; i++)
                {
                    longValue <<= 16;
                    longValue += (long)values[i];
                }
                switch (mt.MeasurementFormat)
                {
                    case "unsigned": break;
                    case "signed":
                        longValue = (int)longValue; break;
                }

                listM.Add(new BufferMeasurementValue()
                {
                    Register = (int)mt.Address,
                    Value = longValue
                });
            }
        }
        catch (Exception err)
        {
            Trace.WriteLine("# 'PollFunction' in 'ElectricityReader: " + err.Message);
        }
    }

}
#endregion
```

Fig. 46 Part of PollFunction method content

The `PollFunction` method iterates over all `MeasurementType` inside machineProtocol1.MeasurementTypeList. This means all the wanted measurement defined in settings.txt.

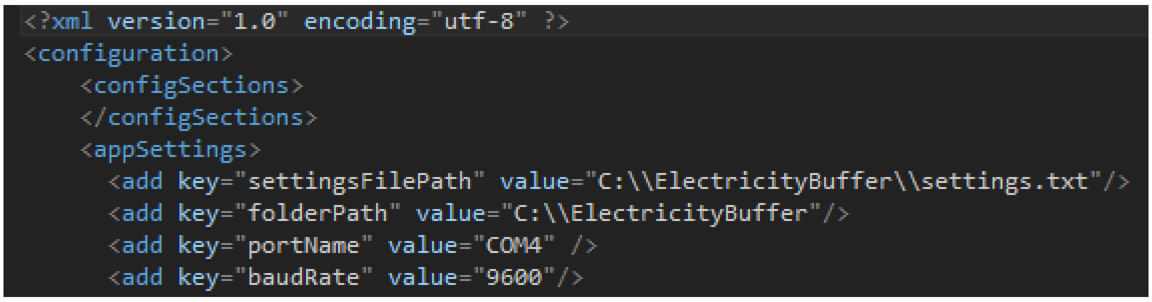
Then `modbus1` object builds message using information in `machineProtocol1` and its current `MeasurementType` element. After that, the message is sent over serial port using `SendFc3` and the reply value is stored in values array.

Then the "`values`" array is converted into number using byte-shifting operation and stored in longValue.

Then depending on its pre-defined format, it will be converted into signed number if needed.

2.6.9 App.config

The file in XML format stores all needed configuration for the application and the file can be changeable. In this way, if there is anything needed to be changed like COM-Port settings or Provide passcode, the user doesn't need to change the project code and compiles the application again.

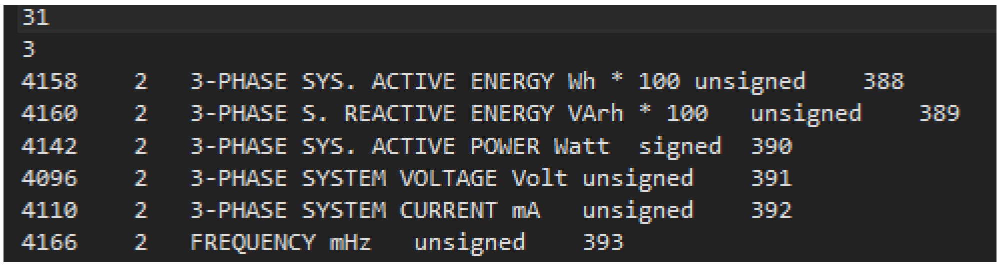


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <appSettings>
      <add key="settingsFilePath" value="C:\\ElectricityBuffer\\settings.txt"/>
      <add key="folderPath" value="C:\\ElectricityBuffer"/>
      <add key="portName" value="COM4" />
      <add key="baudRate" value="9600"/>
```

Fig. 47 Part of App.config content

2.6.10 Settings.txt

Since App.config cannot store list or generic information. Settings.txt text file is used for easy reading through iteration.



```
31
3
4158    2   3-PHASE SYS. ACTIVE ENERGY Wh * 100 unsigned    388
4160    2   3-PHASE S. REACTIVE ENERGY VArh * 100   unsigned    389
4142    2   3-PHASE SYS. ACTIVE POWER Watt  signed  390
4096    2   3-PHASE SYSTEM VOLTAGE Volt unsigned    391
4110    2   3-PHASE SYSTEM CURRENT mA   unsigned    392
4166    2   FREQUENCY mHz   unsigned    393
```

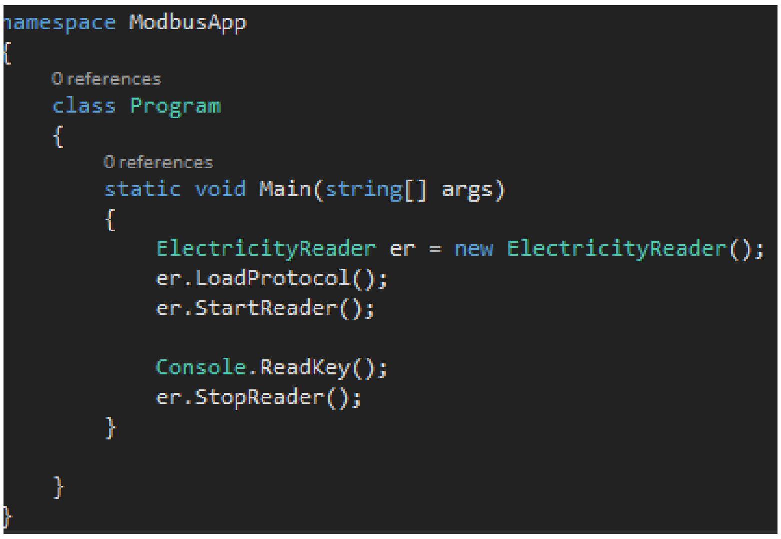
Fig. 48 Part of Settings.txt: 31 is Slave device ID (ABB M2M box), 3 is function code. Each line contains address of start register of measurement in decimal, number of registers, its meaning and unit and corresponding sensor ID on server side.

The format of each measurement is

[Start register address]\t[Number of register to read]\t[Meaning and unit]\t[Match sensor ID on database server]

2.6.11 Main program

The main program is simplified since all functionality is already coded in "ElectricityReader" class. Everything is loaded after starting program. The program reads measurement continuously at interval defined in App.config. The user only need to press any key to stop the program.

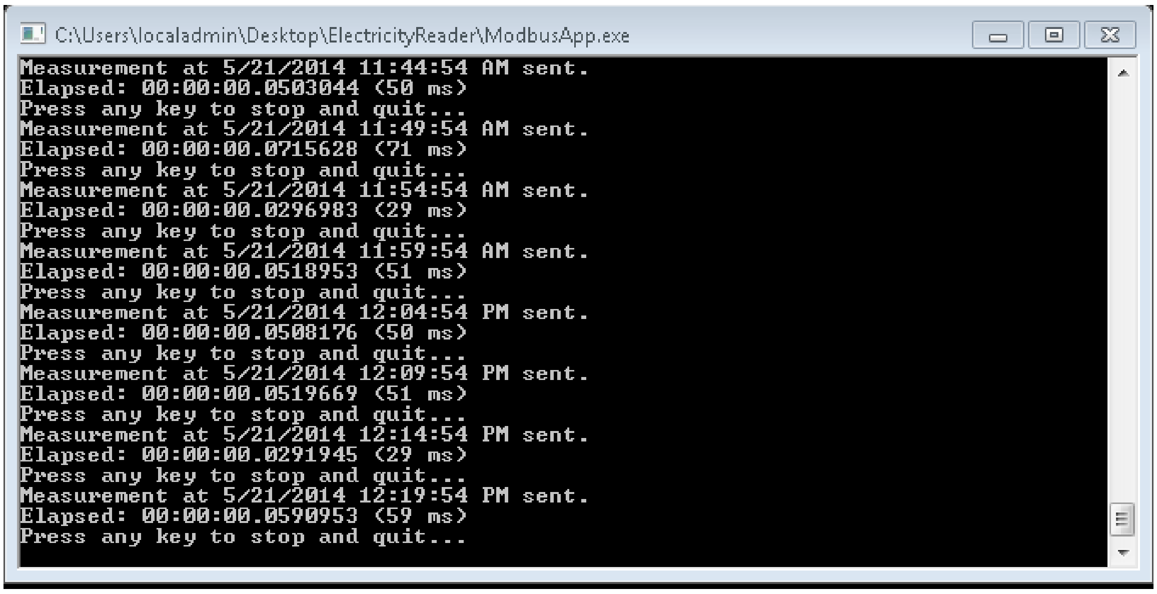


```
namespace ModbusApp
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            ElectricityReader er = new ElectricityReader();
            er.LoadProtocol();
            er.StartReader();

            Console.ReadKey();
            er.StopReader();
        }

    }
}
```

Fig. 49 Main program of ModbusApp

2.7 The result and conclusion

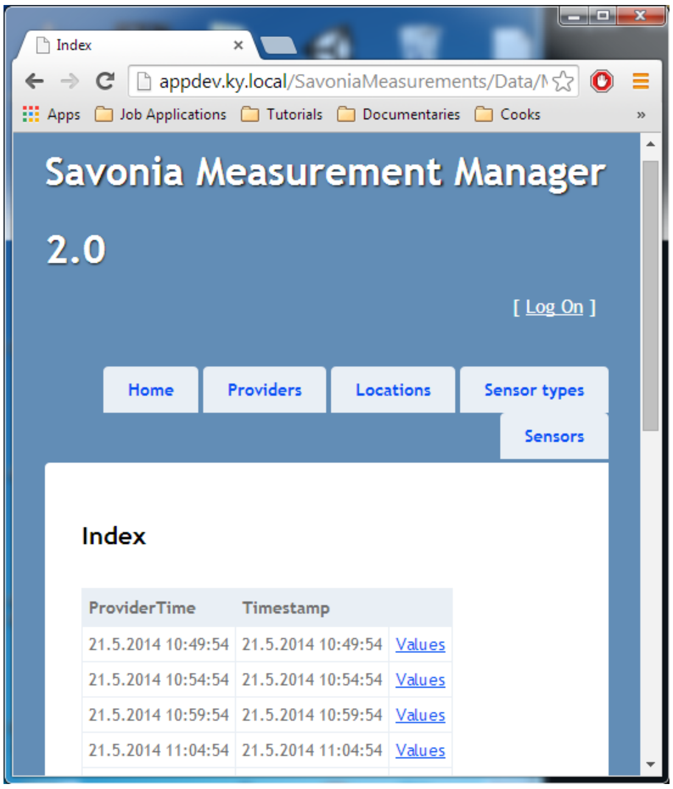
During implementation, we countered various problems. For the first time of testing the reading, I set the reading interval quite low: hundred milliseconds, which means the application queries from the ABB M2M box quite fast. This leads to overflowing problem. The application send too many requests before the slave device can handle and send back reply. The problem makes the PC freezing after some time of running. After that, I set the interval higher from 1 seconds and above. The problem doesn't happen anymore. The PC is left running to collect data for weeks with no error happening.



```
C:\Users\localadmin\Desktop\ElectricityReader\ModbusApp.exe
Measurement at 5/21/2014 11:44:54 AM sent.
Elapsed: 00:00:00.0503044 (50 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 11:49:54 AM sent.
Elapsed: 00:00:00.0715628 (71 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 11:54:54 AM sent.
Elapsed: 00:00:00.0296983 (29 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 11:59:54 AM sent.
Elapsed: 00:00:00.0518953 (51 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 12:04:54 PM sent.
Elapsed: 00:00:00.0508176 (50 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 12:09:54 PM sent.
Elapsed: 00:00:00.0519669 (51 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 12:14:54 PM sent.
Elapsed: 00:00:00.0291945 (29 ms)
Press any key to stop and quit...
Measurement at 5/21/2014 12:19:54 PM sent.
Elapsed: 00:00:00.0590953 (59 ms)
Press any key to stop and quit...
```

Fig. 50 Screenshot of ModbusApp.exe running up to 21.05.2014 with 5 minutes interval. The application is still reliable.

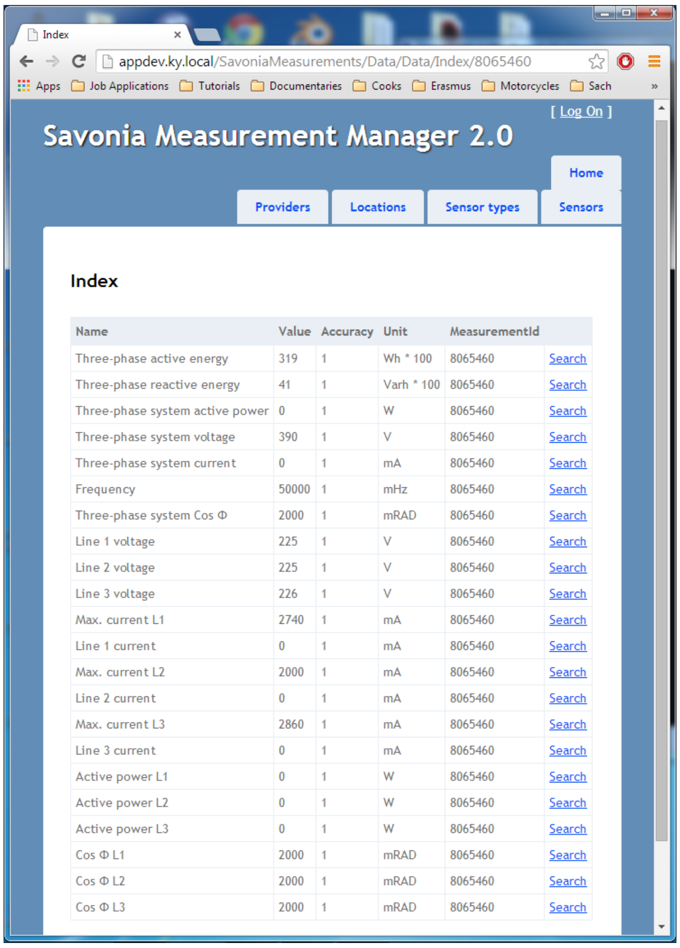
At the time I finish my part, the mobile application client is not finished yet. Therefore, I cannot show any reading from mobile phone client side. However, the person in charge of database server created a simple web client to show measurement stored in database. This client at this moment can only be opened inside Savonia UAS's network. Here are some screenshots of it.



Index

ProviderTime	Timestamp	
21.5.2014 10:49:54	21.5.2014 10:49:54	Values
21.5.2014 10:54:54	21.5.2014 10:54:54	Values
21.5.2014 10:59:54	21.5.2014 10:59:54	Values
21.5.2014 11:04:54	21.5.2014 11:04:54	Values

Fig. 51 Screenshot of testing web client showing list of received measurement packages with their time stamp



Index

Name	Value	Accuracy	Unit	MeasurementId	
Three-phase active energy	319	1	Wh * 100	8065460	Search
Three-phase reactive energy	41	1	Varh * 100	8065460	Search
Three-phase system active power	0	1	W	8065460	Search
Three-phase system voltage	390	1	V	8065460	Search
Three-phase system current	0	1	mA	8065460	Search
Frequency	50000	1	mHz	8065460	Search
Three-phase system Cos Φ	2000	1	mRAD	8065460	Search
Line 1 voltage	225	1	V	8065460	Search
Line 2 voltage	225	1	V	8065460	Search
Line 3 voltage	226	1	V	8065460	Search
Max. current L1	2740	1	mA	8065460	Search
Line 1 current	0	1	mA	8065460	Search
Max. current L2	2000	1	mA	8065460	Search
Line 2 current	0	1	mA	8065460	Search
Max. current L3	2860	1	mA	8065460	Search
Line 3 current	0	1	mA	8065460	Search
Active power L1	0	1	W	8065460	Search
Active power L2	0	1	W	8065460	Search
Active power L3	0	1	W	8065460	Search
Cos Φ L1	2000	1	mRAD	8065460	Search
Cos Φ L2	2000	1	mRAD	8065460	Search
Cos Φ L3	2000	1	mRAD	8065460	Search

Fig. 52 Screenshot of measurement values of one measurement package

Even though, time interval between measurements can be set in real-time (down to one second), it is set to 5 minutes because the capacity of database server may not be enough. 5-minute setting is enough since some of measurement is actually average values over 10 minutes time period.

REFERENCES

[1] Martin P. Bates, “Interfacing PIC Microcontrollers, 2nd Edition”, Chapter 8 Serial Communication, **Publisher:** Newnes, **Pub. Date:** September 18, 2013, **Print ISBN-13:** 978-0-08-099363-8

[2] Jan Axelson, "Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems, 2nd Edition", **Publisher:** Lakeview Research, **Published Date:** 2nd edition (December 1, 2007). **ISBN-13:** 978-1931448062

[3] Casper Yang, “The Secrets of Flow Control in Serial Communication”, Moxa Tech Note, MOXA Inc. Released on Sep 30, 2009

[4] William Buchanan, “Mastering Pascal and Delphi Programming”, **Publisher:** Palgrave Macmillan **Pub. Date** March 23, 1998, **ISBN-13:** 978-0333730072, Chapter 16: RS-232

[5] NorDevX DB-25 Male Connector https://www.nordevx.com/content/db25-male-connector 14.05.2014

[6] Networx DB-25 Female Connector http://www.computercablestore.com/DB25_Female_Solder_Connec_PID49956.aspx 14.05.2014

[7] Commercial Male and Femal DB-9 Connector http://delta-electronic.com/shop/index.php?cPath=50_168&osCsid=8ed4397858dc10786dc5228d986cc675 14.05.2014

[8] Thomas Kugelstadt, “The RS-485 Design Guide-Application Report”, February 2008, Revised May 2008 http://www.ti.com/lit/an/slla272b/slla272b.pdf 14.05.2014

[9] Universal Serial Bus Cables and Connectors Class Document, Revision 2.0, August 2007 http://www.usb.org/developers/devclass_docs/CabConn20.pdf

[10] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips “Universal Serial Bus Specification”, Revision 2.0, April 27 2000

[11] Jan Axelson, “USB Complete, The Developer's Guide 4th Edition”, **Publisher:** Lakeview Research, **Published Date:** 1 June 2009, **ISBN-13:** 978-1931448086

[12] USB 3.0 A & B type connectors example http://www.totalphase.com/support/articles/200349256-USB-Background

[13] modbus.org, MODBUS Application Protocol Specification V1.1b3, http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf 14.05.2014

[14] modbus-IDA.org, “MODBUS Over Serial Line, Specification and Implementation Guide V1.02” http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf 14.05.2014

[15] MODICON, Inc., Industrial Automation Systems, “Modicon Modbus Protocol Reference Guide PI–MBUS–300 Rev. J”, June 1996

[16] ABB, “M2M/DMTME Instruments Communication Protocol, Technical specification V2.0” http://www05.abb.com/global/scot/scot209.nsf/veritydisplay/635fd2b8ca9d770ac12578cb0023448f/$file/2csg445011d0201.pdf 14.05.2014

[17] Farnell, FTDI-USB-RS485-WE-1800-BT http://fi.farnell.com/ftdi/usb-rs485-we-1800-bt/cable-usb-rs485-serial-converter/dp/1740357 14.05.2014

[18] FTDI-USB-RS485-WE-1800-BT datasheet http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_USB_RS485_CABLES.pdf 14.05.2014

[19] Microsoft, Overview of .NET framework http://msdn.microsoft.com/en-us/library/vstudio/zw4w595w(v=vs.110).aspx 14.05.2014

APPENDIX One-“ElectricityReader” Class Source Code

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Timers;
using System.Xml.Serialization;
using System.Configuration;
using ModbusApp.SavoniaMeasurementService;

namespace ModbusApp
{
    class ElectricityReader
    {
        #region PROPERTIES
        public MachineProtocol machineProtocol1
        { set; get; }
        public modbus modbus1
        { set; get; }
        public System.Timers.Timer timer1
        { set; get; }
        public bool isPolling
        { set; get; }
        public bool isProtocolLoaded
        { set; get; }
        public string folderDirectory
        { set; get; }
        public string settingsPath
        { set; get; }
        public int bufferSize
        { set; get; }
        #endregion
        public ElectricityReader()
        {

            machineProtocol1 = new MachineProtocol();
            modbus1 = new modbus();
            timer1 = new System.Timers.Timer();
            isPolling = false;
            isProtocolLoaded = false;
            timer1.Elapsed += new ElapsedEventHandler(timer_Elapsed);
            bufferSize = 0;
            LoadAppConfig();
        }
        private void timer_Elapsed(object sender, ElapsedEventArgs e)
        {
            PollFunction();
        }
        private void LoadAppConfig()
        {
            try
            {
                folderDirectory = ConfigurationManager.AppSettings["folderPath"].ToString();
                settingsPath = ConfigurationManager.AppSettings["settingsFilePath"].ToString();
                machineProtocol1.Portname = ConfigurationManager.AppSettings["portName"].ToString();
                machineProtocol1.Baudrate = Convert.ToInt16(ConfigurationManager.AppSettings["baudRate"]);
                machineProtocol1.DataBits = Convert.ToInt16(ConfigurationManager.AppSettings["dataBits"]);
                machineProtocol1.Parity = ConfigurationManager.AppSettings["parity"].ToString();
                machineProtocol1.StopBits = ConfigurationManager.AppSettings["stopBits"].ToString();
                machineProtocol1.SampleRate =
Convert.ToInt32(ConfigurationManager.AppSettings["sampleRate"]);
                machineProtocol1.MeasurementTypeID =
Convert.ToInt32(ConfigurationManager.AppSettings["MeasurementTypeID"]);
                machineProtocol1.ProviderID =
Convert.ToInt32(ConfigurationManager.AppSettings["ProviderID"]);
                machineProtocol1.ProviderPasscode =
ConfigurationManager.AppSettings["ProviderPasscode"].ToString();
            }
            catch (Exception err)
            {
                Trace.WriteLine("# 'LoadConfig' in 'ElectricityReader': " + err.Message);
            }
        }

        #region METHODS
        public bool StartReader()
        {
            if (isProtocolLoaded)
            {

                try
                {
                    Parity parity = new Parity();
                    StopBits stopbits = new StopBits();
                    switch (machineProtocol1.Parity)
                    {
                        case "none": parity = Parity.None; break;
                        case "odd": parity = Parity.Odd; break;
                        case "even": parity = Parity.Even; break;
                        case "mark": parity = Parity.Mark; break;
                        case "space": parity = Parity.Space; break;
                        default: parity = Parity.None; break;
                    }

                    switch (machineProtocol1.StopBits)
                    {
                        case "none": stopbits = StopBits.None; break;
                        case "one": stopbits = StopBits.One; break;
                        case "onepointfive": stopbits = StopBits.OnePointFive; break;
                        case "two": stopbits = StopBits.Two; break;
                        default: stopbits = StopBits.None; break;
                    }
                    if (modbus1.Open(machineProtocol1.Portname, machineProtocol1.Baudrate,
machineProtocol1.DataBits
                                    , parity, stopbits))
                    {
                        //Set polling flag:
                        isPolling = true;

                        //Start timer using provided values:
                        timer1.AutoReset = true;
                        timer1.Interval = machineProtocol1.SampleRate;
                        timer1.Start();
                        Console.Out.WriteLine("Timer stated...");
                    }
                    //   timer1.Start();
                }
                catch (Exception Err)
                {
                    Trace.WriteLine("# 'StartReader' in ElectricityReader' class: " + Err.Message);
                    return false;
                }
            }
            else
            {
                Trace.WriteLine("# 'StartReader' in 'ElectricityReader' class: " + "Protocol is not loaded
before starting Reader");
                return false;
            }
            return true;
        }
        public bool StopReader()
        {
            try
            {
                isPolling = false;
                timer1.Stop();
                Console.Out.WriteLine("Timer ended...");
                modbus1.Close();
            }
            catch (Exception Err)
            {
                Trace.WriteLine("# 'StopReader' in 'ElectricityReader' class: " + Err.Message);
                return false;
            }
            return true;
        }
        public void LoadProtocol()
        {
            try
            {
                LoadAppConfig();
                System.IO.StreamReader sr = new System.IO.StreamReader(settingsPath);
                machineProtocol1.MachineID = Convert.ToByte(sr.ReadLine());
                machineProtocol1.ReadFunction = Convert.ToByte(sr.ReadLine());
                string temp;
                while (sr.Peek() >= 0)
                {
                    temp = sr.ReadLine();
                    string[] values = temp.Split(new string[] { "\t" },
StringSplitOptions.RemoveEmptyEntries);
                    ushort address = Convert.ToUInt16(values[0]);
                    ushort number = Convert.ToUInt16(values[1]);
                    string meaning = values[2];
                    string format = values[3];
                    int matchSensorID = Convert.ToInt32(values[4]);
                    machineProtocol1.MeasurementTypeList.Add(new MeasurementType(address, number, meaning,
format,matchSensorID));
                }
                isProtocolLoaded = true;
                Console.Out.WriteLine("Settings loaded...");
            }
            catch (Exception err)
            {
                isProtocolLoaded = false;
                Trace.WriteLine("# 'LoadProtocol' in 'ElecityReader': " + err.Message);
            }
        }



        private void PollFunction()
        {
            try
            {
                Dictionary<int, long> l = new Dictionary<int, long>();

                List<BufferMeasurementValue> listM = new List<BufferMeasurementValue>();
                BufferMeasurement bufferM = new BufferMeasurement();

                #region Interact with machines
                foreach (MeasurementType mt in machineProtocol1.MeasurementTypeList)
                {
                    if (machineProtocol1.ReadFunction == 3)
                    {
                        ushort[] values = new ushort[mt.RegisterNumber];
                        try
                        {
                            if (modbus1.SendFc3(machineProtocol1.MachineID, mt.Address, mt.RegisterNumber,
ref values))
                            {
                                long longValue = (long)values[0];
                                for (int i = 1; i < mt.RegisterNumber; i++)
                                {
                                    longValue <<= 16;
                                    longValue += (long)values[i];
                                }
                                switch (mt.MeasurementFormat)
                                {
                                    case "unsigned": break;
                                    case "signed":
                                        //if (longValue > int.MaxValue) longValue -= 4294967296; break;
                                        longValue = (int)longValue; break;
                                }
                                //l.Add((int)mt.Address, longValue);
                                listM.Add(new BufferMeasurementValue()
                                {
                                    Register = (int)mt.Address,
                                    Value = longValue
                                });
                            }
                        }
                        catch (Exception err)
                        {
                            Trace.WriteLine("# 'PollFunction' in 'ElectricityReader: " + err.Message);
                        }
                    }
                }
                #endregion

                bufferM.Timestamp = DateTime.UtcNow;
                bufferM.MachineID = Convert.ToUInt16(machineProtocol1.MachineID);
                bufferM.MeasurementList = listM;
                SendToSever(bufferM);

            }
            catch (Exception err)
            {
                Trace.WriteLine("# 'PollFunction' in 'ElectricityReader': " + err.Message);
            }
        }
        public void SendToSever(BufferMeasurement pBM)
        {
            try
            {
                MeasurementServiceClient ws = new MeasurementServiceClient();
                MeasurementData md;
                md = new MeasurementData();
                md.MeasurementTypeId = machineProtocol1.MeasurementTypeID;
                md.ProviderInfo = new ProviderData();
                md.ProviderInfo.ProviderId = machineProtocol1.ProviderID;
                md.ProviderInfo.ProviderPassCode = machineProtocol1.ProviderPasscode;
                md = ws.GetMeasurementTypeSensors(md);

                md.MeasurementTimeUTC = pBM.Timestamp;
                foreach (var p in machineProtocol1.MeasurementTypeList)
                {
                    md.SensorDatas.First(s => s.Id == p.MatchSensorID).Value =
(double)pBM.MeasurementList.First( s => s.Register== p.Address).Value;
                }


                Stopwatch stopwatch = new Stopwatch();
                stopwatch.Start();
                ModbusApp.SavoniaMeasurementService.MeasurementResponse mr = ws.SaveMeasurement(md);
                stopwatch.Stop();
                ws.Close();
                if (mr.Success)
                {
                    Console.Out.WriteLine("Measurement at " + pBM.Timestamp + " sent.");
                    Console.Out.WriteLine("Elapsed: {0} ({1} ms)", stopwatch.Elapsed,
stopwatch.ElapsedMilliseconds);
                    Console.Out.WriteLine("Press any key to stop and quit...");
                    ReleaseBuffer();
                }
                else
                {
                    Console.Out.WriteLine("Measurement at " + pBM.Timestamp + " sent failed. It will be
stored in buffered.");
                    Console.Out.WriteLine("Press any key to stop and quit...");
                    SaveToBuffer(pBM);
                }
            }
            catch (Exception err)
            {
                Trace.WriteLine("# 'Send' in 'ElectricityReader': " + err.Message);
            }
        }
        public void SaveToBuffer(BufferMeasurement buffer)
        {
            try
            {
                if (!Directory.Exists(folderDirectory))
                    Directory.CreateDirectory(folderDirectory);

                XmlSerializer mySerializer = new XmlSerializer(typeof(BufferMeasurement));
                StreamWriter myWriter = new StreamWriter(folderDirectory + bufferSize.ToString() + ".xml");
                bufferSize++;

                mySerializer.Serialize(myWriter, buffer);
                myWriter.Close();
            }
            catch (Exception err)
            {
                Trace.WriteLine("# 'SaveToBuffer' in 'ElectricityReader': " + err.Message);
            }
        }
        public void ReleaseBuffer()
        {
            MeasurementServiceClient ws = new MeasurementServiceClient();
            MeasurementData md;
            md = new MeasurementData();

            XmlSerializer deSerializer = new XmlSerializer(typeof(BufferMeasurement));
            foreach (string file in Directory.EnumerateFiles(folderDirectory, "*.xml"))
            {
                TextReader textReader = new StreamReader(file);
                BufferMeasurement pBM = (BufferMeasurement)deSerializer.Deserialize(textReader);

                md.MeasurementTypeId = machineProtocol1.MeasurementTypeID;
                md.ProviderInfo = new ProviderData();
                md.ProviderInfo.ProviderId = machineProtocol1.ProviderID;
                md.ProviderInfo.ProviderPassCode = machineProtocol1.ProviderPasscode;
                md = ws.GetMeasurementTypeSensors(md);
                md.MeasurementTimeUTC = pBM.Timestamp;
                foreach (var p in machineProtocol1.MeasurementTypeList)
                {
                    md.SensorDatas.First(s => s.Id == p.MatchSensorID).Value =
(double)pBM.MeasurementList.First(s => s.Register == p.Address).Value;
                }

                Stopwatch stopwatch = new Stopwatch();
                stopwatch.Start();
                ModbusApp.SavoniaMeasurementService.MeasurementResponse mr = ws.SaveMeasurement(md);
                stopwatch.Stop();
                if (mr.Success)
                {
                    Console.Out.WriteLine("Measurement at " + pBM.Timestamp + " sent.");
                    Console.Out.WriteLine("Elapsed: {0} ({1} ms)", stopwatch.Elapsed,
stopwatch.ElapsedMilliseconds);
                    Console.Out.WriteLine("Press any key to stop and quit...");
                    bufferSize--;
                }
                else
                {
                    SaveToBuffer(pBM);
                }
            }
            ws.Close();
        }
        #endregion


    }
}
```

SAVONIA