University of Southern Queensland

Faculty of Health, Engineering and Sciences

Arduino Modbus Simulator

A dissertation submitted by

Ryan Beccarelli

In fulfilment of the requirements of

ENG4112 Research Project

Towards the degree of

Bachelor of Engineering (Computer Systems)

Submitted: October 2016

Abstract

Modbus is an industrial communications protocol used to interconnect control systems and control system input / output equipment such as sensors and transducers. It is platform independent and is commonly used to network control systems from different manufactures. A purpose of the interconnection is to allow data to be stored centrally for analysis and to allow remote control of package control systems by a master system. While Modbus is commonly used, there are a limited number of diagnostic tools and solutions for use by technicians.

This dissertation documents the software and hardware design of an Arduino microcontroller based Modbus simulator to give end users such as technicians and engineers a new tool to use for commissioning and troubleshooting Modbus networks. The outcome of this project is a working Arduino Modbus Simulator prototype, that has been successfully tested with industrial control systems.

Benefits delivered by the project can be summarised into three areas being:

1. Reducing the Mean Time to Repair of a Modbus serial communication link
2. Ergonomic and simple to use alternative to computer based systems
3. Competitive open source solution to propriety hardware and software

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

R. Beccarelli

61003049

Acknowledgements

A special thanks goes to my supervisors, Mrs Catherine Hills and A/Prof Alexander Kist, for their valuable time in providing guidance during the execution of this project.

I would also like to thank all the USQ faculty that have guided me over the years of my study. It has been a challenging time for me and I appreciate all help that has been given.

Finally, thank you to Caitlin and Alex, for your patience and always supporting me on this journey.

RYAN BECCARELLI

University of Southern Queensland

September 2016

Contents

List of Figures

List of Tables

Nomenclature

ADU	Application Data Unit
ASCII	American Standard Code for Information Interchange
CRC	Cyclical Redundancy Check
DCS	Distributed Control System
FPSO	Floating Production Storage and Offtake
HART	Highway Addressable Remote Transducer
LCD	Liquid Crystal Display
OPC	Object Linking and Embedding for Process Control
PCS	Process Control System
PLC	Programmable Logic Controller
SIS	Safety Instrumented System
TCM	Tricon Communication Module
TCP/IP	Transmission Control Protocol / Internet Protocol
TMR	Triple Modular Redundant
TTL	Transistor to Transistor Logic
UART	Universal Asynchronous Receiver Transmitter

1. Introduction

This dissertation documents the hardware and software solution for the Arduino Modbus Simulator. The purpose of this project is to develop and the prove the concept of a handheld Modbus serial network test tool based on the Arduino open source development platform. This dissertation provides a literature review of what Modbus is, where and how it is used and why there is a need for this project. Some of the information contained in this dissertation was previously submitted as an unpublished works assignment by the author Beccarelli, R 2015, 'ENG4110 Engineering Research Methodology Assignment 2', Coursework assignment, University of Southern Queensland.

The motivation for completing this project was driven by the author's experience where an industrial control system Modbus protocol network had failed, causing both economic losses and safety concerns. There was clearly a lack of test equipment available for Modbus in comparison to what was freely available for other control system apparatus.

The aim of the project is to design and prototype an Arduino based Modbus simulator to give end users such as technicians and engineers a new tool to use for commissioning and troubleshooting Modbus networks.

Broadly the hardware and software development must satisfy the general objectives being:

- Hardware to be small and compact
- Hardware to have easy to identify communication leads and connectors that are simple to use
- Software interface is a simple design with clear instructions with the user controls
- Robust and reliable in both software and hardware aspects
- The hardware and software application to successfully work with many different Modbus equipment manufacturers

The software and hardware solution must meet the following requirements in order to deliver a successful outcome for the project:

- Hardware solution will use RS-485 as the electrical standard. RS-485 has been chosen as 2 wire solution is very common in industry, is a multi-drop connection making it very flexible for measuring voltages and testing, and available test equipment is compatible with RS-485.
- The software solution will use the Modbus Remote Terminal Unit (RTU) protocol. RTU has been chosen as it is the more commonly used serial interface than the ASCII alternative.
- The software solution will be a Modbus Master. Master has been chosen over Slave as the majority of the target devices for this application are of the Slave type. For example, one Distributed Control System (DCS) Master communicates to many Slaves.

- The software solution must be able to use Function Code 03 Holding Register to write to Slave in the Double Integer format
- The software solution must be able to use Function Code 04 Input Register to read from Slave in the Double Integer format.
- The software solution must be able to use Function Code 02 Input Status to read from Slave in the Boolean format
- The software solution must be able to use Function Code 01 Coil Status to write to Slave in the Boolean format
- The software solution must perform a Cyclical Redundancy Check as its error checking mechanism.

1.1. Modbus and Process Control Systems

Mackay (2004) describes Modbus as an industrial communications protocol that ‘has become the de facto standard in multi-vendor integration’. Modbus was invented by Modicon, now Schneider Electric, in 1979 but it still remains today as the protocol universally used to interconnect devices from different manufacturers. This is stated not only by many texts but a product search of the main automation companies such as Yokogawa, Honeywell, Emerson and Allen Bradley will reveal they offer their own propriety communication protocols for communication between its own products and they also all offer Modbus as a way of connecting their products to other manufacturers.

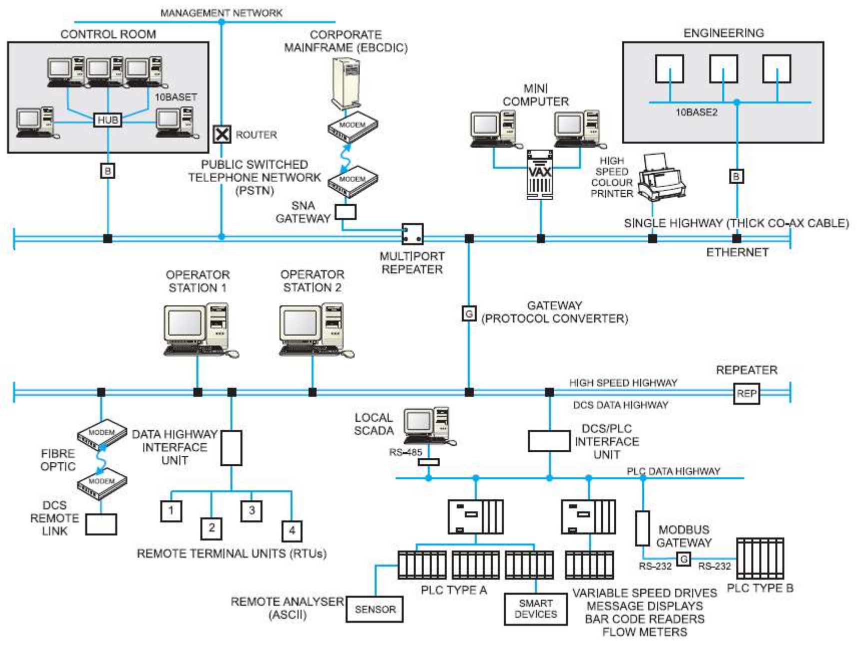


Figure 1.1: Graphical representation of data communication in industrial control system (Mackay 2004)

As represented in Figure 1.1, the integrated Process Control System can be broken into various systems. Modern manufacturing and processing facilities are controlled by an integrated Process Control System that provides remote measurement, remote transducer actuation and with control from a centralised location. The control functions can be automatic or with manual operator interaction via a computer interface.

Distributed Control System (DCS) – used as the primary process control and data acquisition system. Nodes of input and output modules controlled by central processing units (CPU) are interconnected via industrial communications network, generally a proprietary protocol.

Programmable Logic Controllers (PLC) – small and compact, with fast processing and control of input and outputs. The central processor executes the program code for control locally. Generally used for utility packages such as gas turbine controls or power management systems. The PLC is interconnected to the DCS with a communications link to allow data to be viewed in the central control room or stored in a data acquisition database for future analytic purposes.

As the DCS and PLC equipment are normally different vendors, Modbus is the protocol often used to bridge the systems, as most vendors support this communication system.

Instrumentation can be described as the sensor device or final element device. Sensors read temperature, flow, pressure, density, voltage and many other measurements. Final elements are valve actuators and positioners, motor controllers, compressor controllers, heater element controllers and many other types. Analogue sensors work by outputting a 4-20mA current signal that is proportional to the measured signal. A smart instrument transmits this data digitally, allowing for more information to be transmitted. Smart instruments connect into a PLC or DCS by what is known as a fieldbus, such as HART, Foundation Fieldbus, Profibus, DeviceNet and many other types.

Research has demonstrated that the tools available to the technicians and engineers to troubleshoot and test are not readily available or practical. For example, a field located Programmable Logic Controller (PLC) will connect via Modbus into the main facility Distributed Control System (PCS). When troubleshooting or commissioning the link is it typical for a divide and conquer approach where the PLC is disconnected and a laptop running Modbus simulation software is connected into the loop. However, a laptop is large and bulky, requires external electrical interfaces and special cables and generally has to be setup every time it is used.

In comparison for commissioning and troubleshooting smart instrumentation networks such as HART and Fieldbus, manufacturers such as Emerson offer hand held communicators. Research has failed to find such a device for performing diagnostics on Modbus protocol.

2. Literature Review and Modbus Software Hardware Review

This section is a review of available literature for the following key concepts.

1. Modbus Introduction
2. Modbus Serial Transport Layer
3. Modbus Ethernet Transport Layer
4. Modbus Datalink Layer
5. Modbus Application Protocol
6. Error Checking
7. Survey of Existing Installations
8. Proprietary Software Options
9. Open Source Software
10. Hardware Options
11. Software Testing Methodology

2.1. Modbus Introduction

As previously discussed, Schneider Electric introduced the Modbus protocol to the market in 1979. In continuing support and development of Modbus, Schneider created a website as the central information and reference point for everything related to the protocol.

Schneider supported and maintained the Modbus site in the past but understanding the important role Modbus has to play in the market and its use by vendors who are competitors, Schneider Electric assisted in the development of an independent developer and user community organization: The Modbus Organization.

The Modbus Organisation openly provides the Modbus Specification and Implementation Guides for use by developers. This allows developers to build the hardware and implement the software on any platform they choose, independent of using products from Schneider Electric.

The guiding documents are

- Modbus over Serial Line Specification and Implementation Guide V1.02 (2006)
- Modbus Application Protocol Specification V1.1b3 (2012)
- Modbus Messaging on TCP/IP Implementation Guide V1.0b (2006)

The Modbus Organisation also provides a test specification that defines the tests that are to be used and successfully completed to declare conformance to the Modbus protocol specifications

- Conformance Test Specification for Modbus TCP Version 3.0 (2006)

Modbus is an application layer messaging protocol, positioned at level 7 of the OSI model, which provides client/server communication between devices connected on different types of buses or networks.

Modbus is a request/reply protocol and offers services specified by function codes. Modbus function codes are elements of Modbus request/reply data.

Referencing Figure 2.1 and the Modbus Application Protocol Specification V1.1b3 (2012), Modbus can be defined as "an application layer messaging protocol for client/server communication between devices connected on different types of buses or networks"

It can be implemented using TCP/IP over Ethernet or as serial transmission over media including copper wires, fibre and radio.

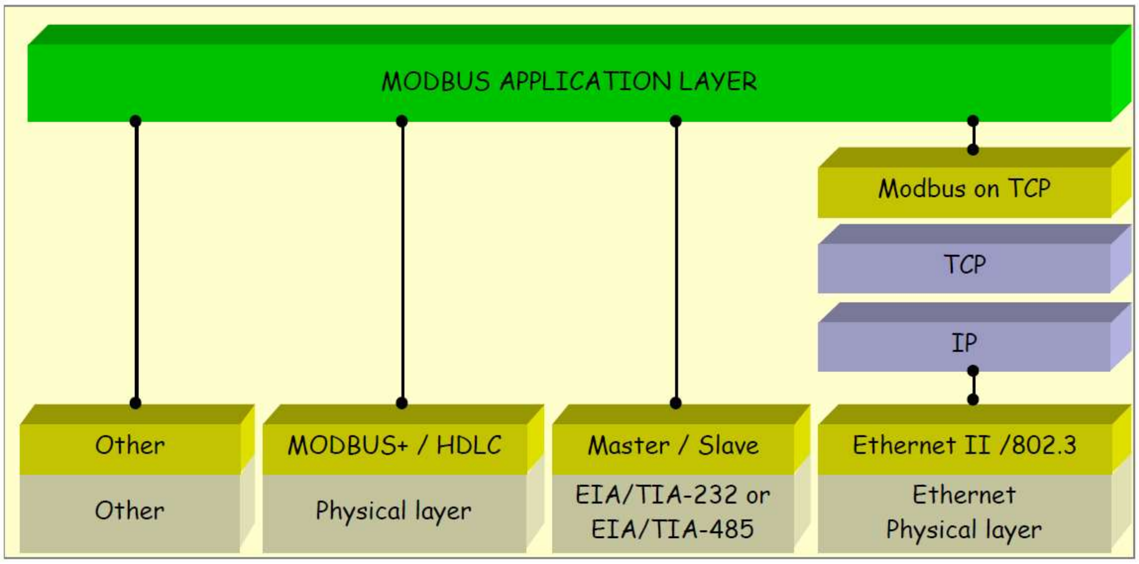
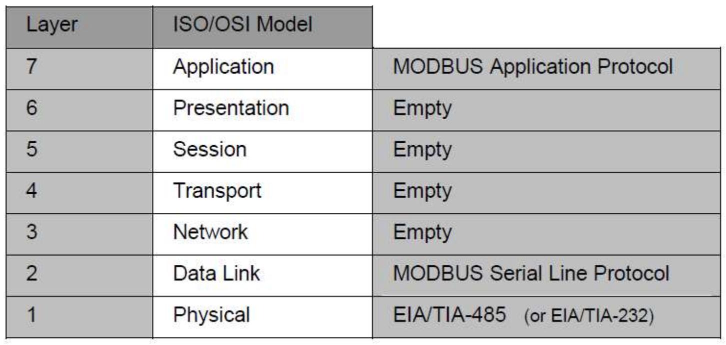


Figure 2.1: Modbus communication stack (Modbus Organsiation 2015)

Figure 2.2 gives a general representation of MODBUS serial communication stack compared to the 7 layers of the OSI model.



Layer	ISO/OSI Model	
7	Application	MODBUS Application Protocol
6	Presentation	Empty
5	Session	Empty
4	Transport	Empty
3	Network	Empty
2	Data Link	MODBUS Serial Line Protocol
1	Physical	EIA/TIA-485 (or EIA/TIA-232)

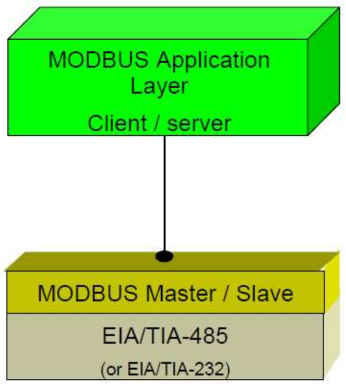


Figure 2.2: Modbus Protocol – Serial Implementation and OSI stack (Modbus Organisation 2015)

2.2. Modbus Physical Layer – Serial

Serial data is transmitted electrically by the following two methods, RS-232 or RS-485. RS-232 stands for Recommended Standard number 232 and RS-485 is Recommended Standard number 485. These standards are by the Telecommunications Industry Association (TIA) and can also be referred to as TIA-232 and TIA-485. The reference documents used for the literature review of serial ports including both RS-232 and RDS-485 are

- Serial Port Complete (Axelson 1998)
- Serial port and microcontrollers (Niemirowski 2013)
- Practical Troubleshooting and Problem Solving of Modbus & Modbus Plus Protocols, a course book produced by IDC Technologies

2.2.1. RS-232

RS-232 transports signals as bits by creating a voltage potential across the transmit wire and the ground wire. A receiver device measures the potential and stores the signal as a 1 or 0.
RS-232 is a single sender, single receiver communication method. It is not possible without the addition of further electronic circuits to drive signals to multiple destinations.

While RS-232 designates there are 25 lines, wires to carry signals, most interfaces rarely support more than 9. Axelson (1998) states there are 3 essential signals for 2-way RS-232 communication being

- Transmit Data also called TX
- Receive Data also called RX
- Signal Ground also called GND

Niemirowski (2013) defines voltage characteristics of RS-232 being

- Receiver input sensitivity +3 volts to – 3 volts
- Receiver input range +15 volts to -15 volts

- Maximum Transmit driver output +25 volts to -25 volts
- Minimum Transmit driver output +5 volts to -5 volts

A logic 0 Space is a voltage +5 volts to +15 volts on the Transmit side to +3 volts to +15 volts on the Receive side.

A logic 1 Mark is a voltage -5 volts to -15 volts on Transmit side to -3 volts to -15 volts on the Receiver side.

2.2.2. RS-485

RS-485 has an advantage over RS-232 in that it has the ability to drive electrical signals to multiple destinations. RS-485 has another advantage as by not using the electrical common as a reference signal, it has a higher tolerance to electrical noise. RS-485 creates a voltage potential across a pair of wires and therefore any electrical noise affects both wires equally. This means the potential doesn’t change.

Niemirowski (2013) defines voltage characteristics of RS-485 being

- Receiver input sensitivity +200 millivolts to – 200 millivolts
- Receiver input range +12 volts to -7 volts
- Maximum Transmit driver output +12 volts to -7 volts
- Minimum Transmit driver output +1.5 volts to -1.5 volts

In RS-485 a pair of wires transmits a small voltage and the receiver looks at a differential. If wire A is a voltage of 200mv greater than wire B, then it is considered a true condition (A logic 1 Mark). If wire B voltage is 200mv greater than wire A, then it is a false condition (A logic 0 Space).

2.3. Ethernet

Modbus communications over Ethernet is known as Modbus TCP. It uses the Modbus Application Protocol to produce a Modbus message that contained within a TCP packet, which is contained in a IP packet. Ethernet is just the electrical specification used to transmit the signal. This is explained by Practical Troubleshooting and Problem Solving of Modbus & Modbus Plus Protocols, a course book produced by IDC Technologies.

2.4. Modbus Data Link Layer

The Modbus over Serial Line Specification and Implementation Guide V1.02 (2006) specifies Modbus serial protocol is a Master to Slave relationship as per Figure 2.3 with the following criteria

- There is only one master connected onto the bus
- There can be 1 to a maximum 247 Slaves nodes connected to the bus.
- Only the Master can initiate communication

- Slave nodes will only transmit data when requested from the master node.
- A Slave node will never communicate to another Slave node
- The Master node executes one Modbus transaction at a time

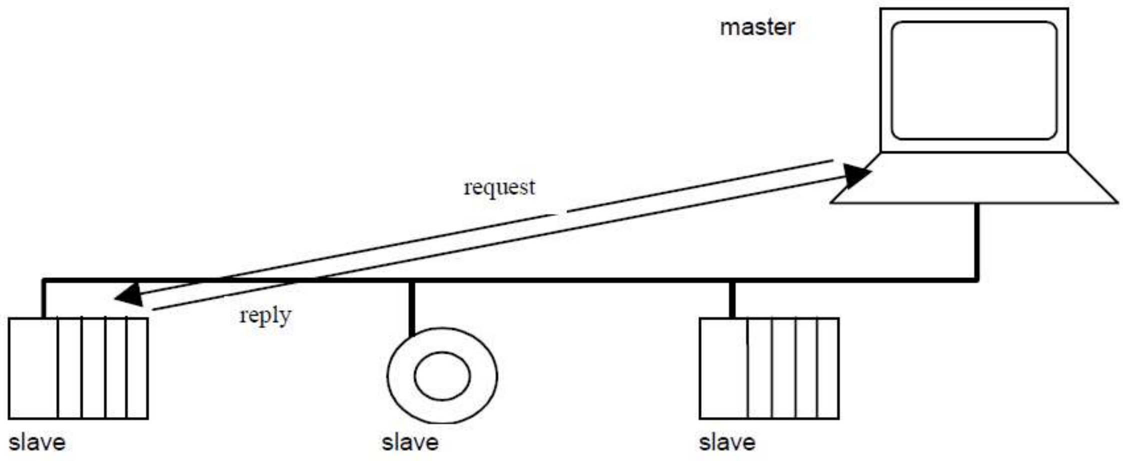


Figure 2.3: Modbus Master requesting data from Slave (Modbus Organisation 2015)

2.5. Modbus Application Protocol

The Modbus Application Protocol Specification V1.1b3 (2012) states that at the application layer compromises of 4 main fields being

- Addressing. This is the device address of the slave device
- Function Code. This specifies what data transaction the Master requests the Slave to perform
- Addressing rules specific that register data is addressed 0 to 65535 with four data types. The data types are Discrete Inputs, Coils, Input Registers and Holding Registers. Each data type has a unique range in the 0 to 65535 range.
- Error checking is executed by performing Cyclical Redundancy Checking (CRC) on the message contents. The sending device calculates the CRC based on the contents of the message and appends value to the message. The receiving device performs its own calculation and compares result to what is in the CRC field.

These four fields are known as the Application Data Unit (ADU).

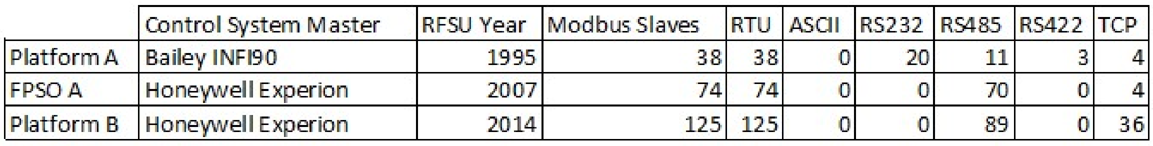
The two methods for encoding Modbus message are RTU and ASCII. An encoding mechanism is the rule for how bit patterns are created from control and data bits to form a packet. RTU encoding sends the values as binary with the most significant bit first and the least significant bit last. ASCII encoding sends the ASCII value down the wire instead of using binary.

2.6. Error Checking

Axelson (1998) states that a receiver uses error checking to verify that all data arrived from the transmitter correctly. This can be done by sending redundant data or applying error checking bytes in the message. Modbus RTU uses the Cyclic Redundancy Code (CRC) method, which applies complex math to determine the data integrity. A simple way to describe a CRC calculation is to take the message data, convert to binary and divide by a particular value. The remainder returned is stored as the CRC. The CRC is added to end of message and transmitted. The receiver takes the message, performs the same mathematical division and compares the result against the CRC. If both are the same, the message was not corrupted in transmission.

2.7. Survey of Existing Installations

A survey of 3 different oil & gas facilities Process Control Systems (PCS) has been conducted. The 3 facilities were constructed and commissioned over a 21-year time horizon. The survey was a review against the PCS architecture drawings to provide insight to how many and what type of Modbus links with the results displayed in Figure 2.4.



	Control System Master	RFSU Year	Modbus Slaves	RTU	ASCII	RS232	RS485	RS422	TCP
Platform A	Bailey INFI90	1995	38	38	0	20	11	3	4
FPSO A	Honeywell Experion	2007	74	74	0	0	70	0	4
Platform B	Honeywell Experion	2014	125	125	0	0	89	0	36

Figure 2.4: Types and Number of Modbus Slaves in oil & gas installations

The results from the sample group highlight some key points for consideration being

- All slaves are using RTU as the Message Encoding format
- The number of devices that are connected via communications link to Process Control System is growing
- RS-485 is the dominant physical connection but Ethernet (TCP/IP) is gaining momentum. RS-232 is non-existent in the new facilities

2.8. Proprietary Software

There are a variety of Modbus software simulation packages available. They vary from Applications running on a Windows platform, Object Linking and Embedding for Process Control (OPC) software and software running on a hardware gateway. Some examples that been reviewed include

- Wintech ModScan32
- WinTech ModSim32
- Matrikon OPC for Modbus
- Real Time Automation Source Code Stacks

ModScan32 is an application that runs on a Windows PC and acts as a Modbus Master device. It can connect to a slave using serial or Ethernet communication reading and writing to data points in either RTU or ASCII message encoding. WinTech sells a single user license for ModScan32 Pro software for US$109.95.

ModSim32 is an application that runs on a Windows PC and simulates as a Modbus Slave device. It is an application that defines blocks of data points that can be read and written by a Modbus Master. It can connect by serial and Ethernet and data can be in RTU or ASCII format. WinTech sells a single user license for ModSim32 for US$84.95.

Matrikon offers an OPC server software application that can be a Master or Slave device, connected by serial or Ethernet. Figure 2.5 displays a hierarchy tree configured with a Master Ethernet node, Master serial node and a Slave serial node.

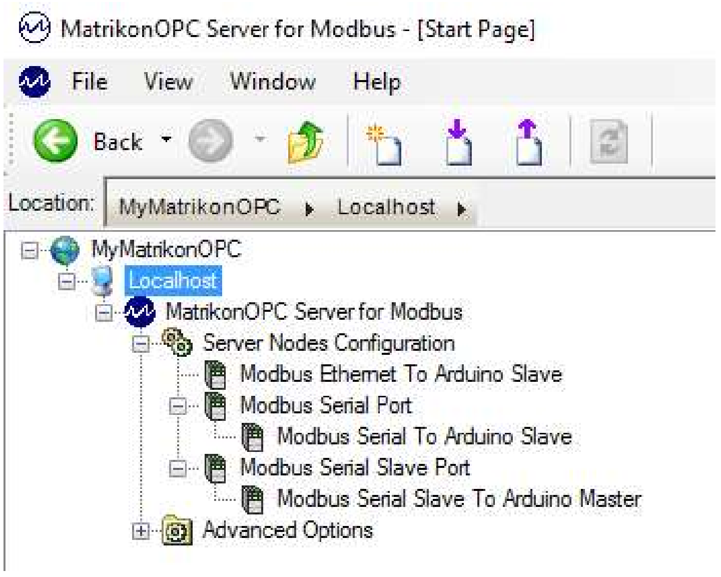

MatrikonOPC Server for Modbus - [Start Page]
File
View
Window
Help
Back
Location:
MyMatrikonOPC
Localhost
MyMatrikonOPC
Localhost
MatrikonOPC Server for Modbus
Server Nodes Configuration
Modbus Ethernet To Arduino Slave
Modbus Serial Port
Modbus Serial To Arduino Slave
Modbus Serial Slave Port
Modbus Serial Slave To Arduino Master
Advanced Options

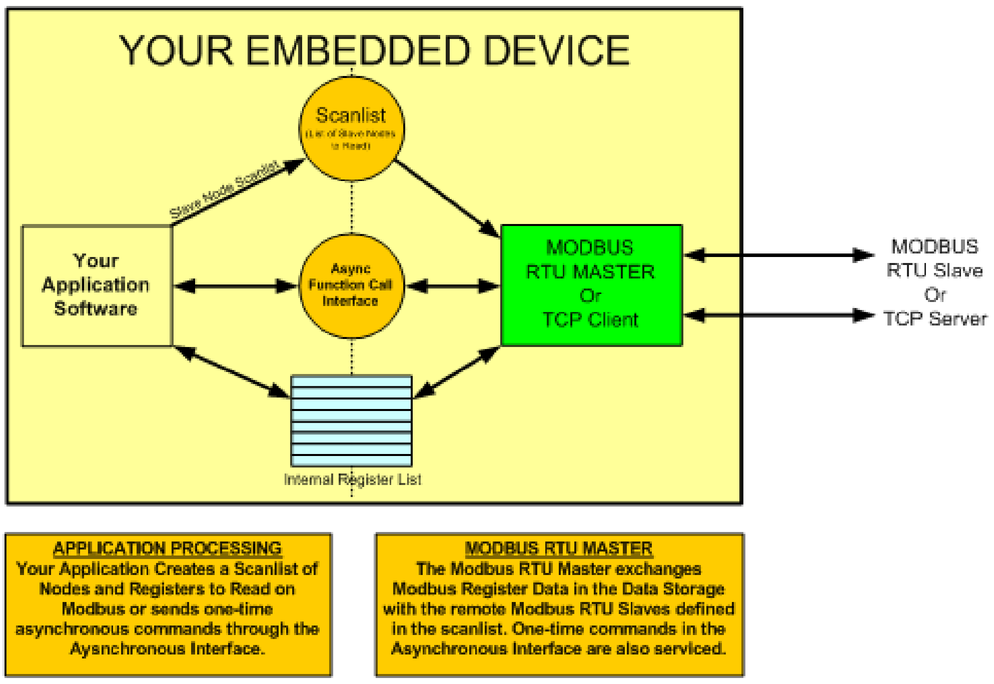
Figure 2.5: MatrikonOPC Server for Modbus (Matrikon 2016)

Matrikon offers this product as 30 day free trial with the license cost for continuing use US$2000 for unlimited tags. This software requires a Windows Operating System and associated hardware to run.

Real Time Automation offers Modbus TCP Client, Modbus TCP Server, Modbus RTU Master and Modbus RTU Slave source code stacks. These products are written in ANSI C. They can be integrated into any hardware platform with any compiler or any operating system. There are two main integration requirements for a Modbus RTU

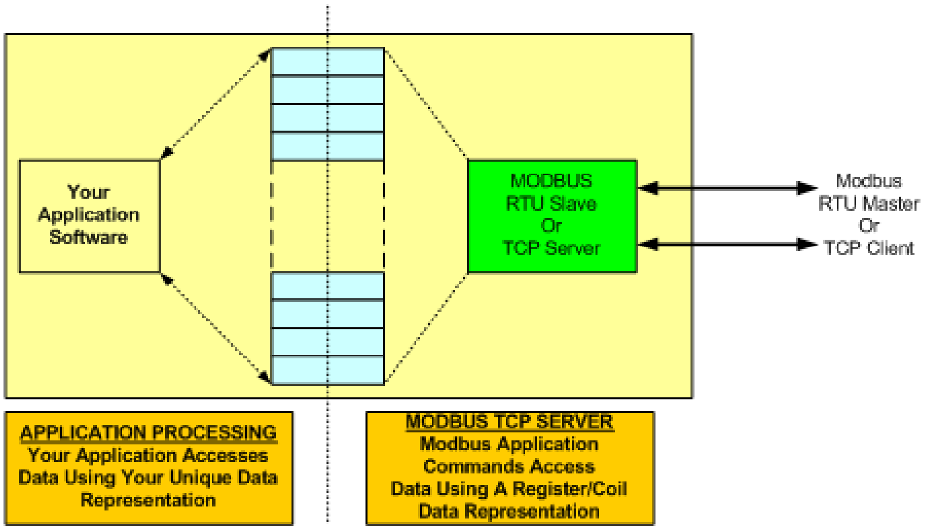
- Connect the low level get or send character of the code stack with the microcontroller UART
- Provide timer functionality so that the source code software can detect timeouts

Figure 2.6 provides an overview of the Real Time Automation Modbus Master Interface while Figure 2.7 depicts a similar structure for the Modbus RTU slave. The Modbus RTU Slave Royalty Free Source Code Stack costs US$1,295.00 and the Modbus RTU Master Royalty Free Source Code Stack costs US$1,795.00.



YOUR EMBEDDED DEVICE
Scanlist
(List of Slave Nodes to Read)
Slave Node Scanlist
Your Application Software
Async Function Call Interface
MODBUS RTU MASTER Or TCP Client
MODBUS RTU Slave Or TCP Server
Internal Register List
APPLICATION PROCESSING
Your Application Creates a Scanlist of Nodes and Registers to Read on Modbus or sends one-time asynchronous commands through the Aysnchronous Interface.
MODBUS RTU MASTER
The Modbus RTU Master exchanges Modbus Register Data in the Data Storage with the remote Modbus RTU Slaves defined in the scanlist. One-time commands in the Asynchronous Interface are also serviced.

Figure 2.6: Modbus RTU Master Interface (Real Time Automation 2015)



Your Application Software
MODBUS RTU Slave Or TCP Server
Modbus RTU Master Or TCP Client
APPLICATION PROCESSING
Your Application Accesses Data Using Your Unique Data Representation
MODBUS TCP SERVER
Modbus Application Commands Access Data Using A Register/Coil Data Representation

Figure 2.7: Modbus RTU Slave Interface (Real Time Automation 2015)

2.9. Open Source Software

A review of open source software options freely available on the internet has been conducted focused on searching for Modbus Master and Modbus Slave examples. The results returned can be grouped into the hardware platforms required to run the software as is without porting. The hardware platforms being

- Arduino
- Raspberry PI
- Microchip PIC
- Windows Applications

By completing a review of the Arduino website code examples for Communications, an example Modbus Master titled "ModbusMaster" is available. This library has been released under the GNU General Public License. A review of the features of this code indicates it has the ability to use RTU protocol over RS-232 or RS-485 and has implemented the following functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs
- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers
- Function Code 5 Write Single Coils
- Function Code 6 Write Single Registers
- Function Code 15 Write Multiple Coils
- Function Code 16 Write Multiple Registers

Another library listed on the Arduino example page is titled "Modbus-arduino". A review of this library indicates it is a Modbus Slave, supporting RTU protocol over serial RS-232 or RS-485 and also TCP/IP via Ethernet or WiFi. It supports the following Modbus functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs
- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers
- Function Code 5 Write Single Coils
- Function Code 6 Write Single Registers
- Function Code 15 Write Multiple Coils

- Function Code 16 Write Multiple Registers

A search of the Arduino forum provides another open source library “simple-modbus”. This library offers both a Master Option, SimpleModbusMaster, and a Slave option, SimpleModbusSlave. The Master library implements the following Modbus functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs
- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers
- Function Code 15 Write Multiple Coils
- Function Code 16 Write Multiple Registers

The slave library implements the following Modbus functions

- Function Code 3 Read Holding Registers
- Function Code 16 Write Multiple Registers

A further internet search reveals an option for a Modbus slave using the RTU protocol “Arduino-modbus-slave”. This library supports serial communications and implements the following Modbus functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs
- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers

An example for a Raspberry PI Modbus Master can be found at the program-plc website. This example implements a Modbus Master using RTU protocol over RS-232 serial communication. It implemtns the following Modbus functions

- Function Code 3 Read Holding Registers
- Function Code 16 Write Multiple Registers

An example library for running on the Microchip PIC hardware platform called “freemodbus-v1.4”. This is a comprehensive library that offers a Modbus Master or Modbus Slave option, can use either RTU or ASCII protocol and can be implemented over RS-232 or RS-485 serial and TC/IP. It implements the following functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs

- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers
- Function Code 5 Write Single Coils
- Function Code 6 Write Single Registers
- Function Code 15 Write Multiple Coils
- Function Code 16 Write Multiple Registers

A Windows application that is available under the General Public License for use is QModBus. This application uses the PC serial port to connect to a Modbus Slave and can send commands for both RTU and ASCII protocols implementing Modbus functions

- Function Code 1 Read Coils
- Function Code 2 Read Discrete Inputs
- Function Code 3 Read Holding Registers
- Function Code 4 Read Input Registers
- Function Code 5 Write Single Coils
- Function Code 6 Write Single Registers
- Function Code 15 Write Multiple Coils
- Function Code 16 Write Multiple Registers

2.10. Hardware Options

Williams (2014) defines a microcontroller has at its core a processor that reads instructions from memory, performs mathematics on a arithmetic logic unit and stores variables in random access memory all while running a program. A microcontroller can be programmed using any of a wide variety of programming languages such as C, Java, Python and many more.

An Arduino, as declared by Blum (2013) is a microcontroller development platform that has been paired with easy to use programming language, based on C. There are many models of Arduino and because it is open source hardware, schematics, source code and design files are available to everybody.

The functionality of an Arduino is easily expanded by the addition of shield. A shield is an electronic circuit that plugs into the pins of the Arduino board to enhance capability by providing more complex circuitry than what is provided on the basic board.

Hardware to be tested include the Arduino Uno that has one on-board Serial port with the option to create virtual software ports if additional ports required.

The Arduino Mega has three on board Serial ports to use and will also be used during the development process.

2.11. Software Testing Methodology

There are many books dedicated to the subject of software testing. Software Testing - A Craftsman's Approach (Jorgensen 2014) has been selected as the primary resource for this project. The book is broken into 3 parts, with part one focusing on Mathematical Context, part two on Unit Testing, and part three on Beyond Unit Testing. Topics of focus useful for the development of a testing methodology include

- Basic Definitions
- Test Cases
- Identifying Test Cases
- Specification Based Testing
- Code Based testing
- Levels of Testing specifying V-Model

A further reference is Testing Computer Software (Kaner, Faulk and Nguyen 1999). This book provides a good overview of testing fundamentals such as the objectives and limits of testing, software errors and the reporting and analysing of bugs. Section 2 of this reference provides a good introduction to specific testing skills including

- Problem Tracking
- Test Case Design
- Testing User Manuals
- Testing Tools
- Test Planning and Test Documentation

A third software testing reference is Software Testing (Patton 2006). This text provides an introduction to the testing process with specific chapter that focuses on planning including

- Goals of Test Planning
- High level expectations
- Definitions
- Defining what will and won't be tested
- Test phases

- Test strategy
- Resource requirements
- Test schedule
- Test cases
- Bug reporting
- Metrics and Statistics
- Risks and Issues

This text provides a useful guide to measuring success of testing, by providing guidance on how to use the data contained in a bug tracking database and example metrics to use in project level testing.

3. Analysis of Existing System

The purpose of this chapter is to review the functions of an existing control system using Modbus serial communication interfaces. This will be achieved by presenting an overview of the test control systems, followed by a review of how Modbus communication is implemented. In order to consider hardware and software designs for the Arduino Modbus Simulator, it is necessary to be familiar and understand how Modbus is implemented on an industrial platform.

1. Invensys Triconex
2. Triconex Slave
3. Triconex Master
4. Allen Bradley SLC500
5. SLC500 Slave
6. Signal Interfaces
7. Test Systems

3.1. Invensys Triconex

The Triconex Planning and Installation Guide describes the Tricon controller as a state of the art programmable logic and process controller that provides a high level of system fault tolerance. The guide states the system has the following features

- Triple modular redundant (TMR) architecture
- Ability to withstand harsh industrial environments
- Allows for online input / output module replacement without disturbing field wiring
- Supports 118 different input / output modules including communication modules
- Has integral online diagnostics capabilities

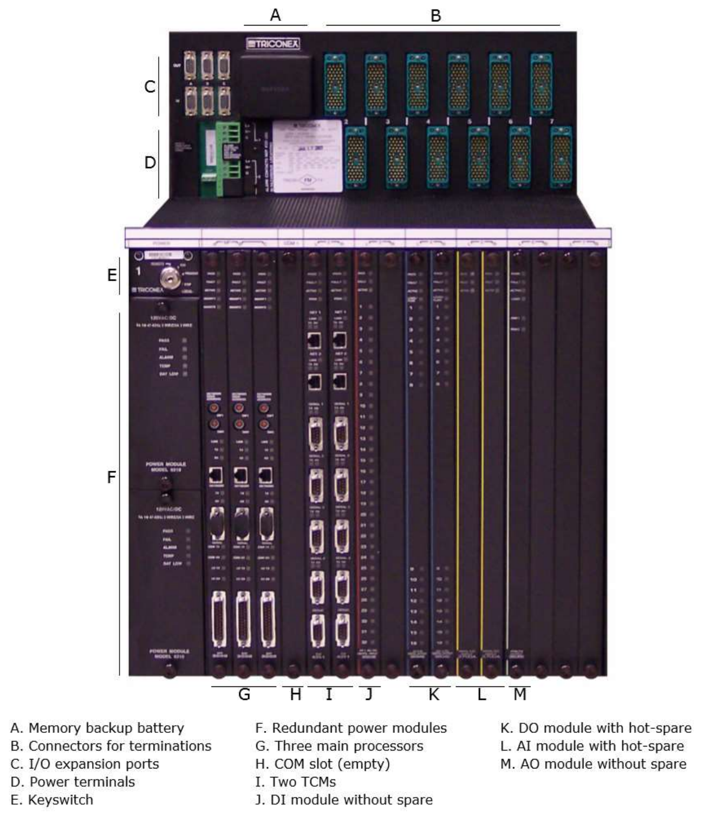

A
B
C
D
E
F
G
H
I
J
K
L
M
A. Memory backup battery
B. Connectors for terminations
C. I/O expansion ports
D. Power terminals
E. Keyswitch
F. Redundant power modules
G. Three main processors
H. COM slot (empty)
I. Two TCMs
J. DI module without spare
K. DO module with hot-spare
L. AI module with hot-spare
M. AO module without spare

Figure 3.1 Typical Tricon Main Chassis (Invensys 2015)

Triconex controllers are used as the logic solver in a Safety Instrumented System due to the high equipment reliability that comes with the hardware being Triple Modular Redundant and having onboard diagnostic capability. They are commonly used in process safety applications such as petrochemical refining, upstream oil & gas processing and nuclear reactor safeguarding.

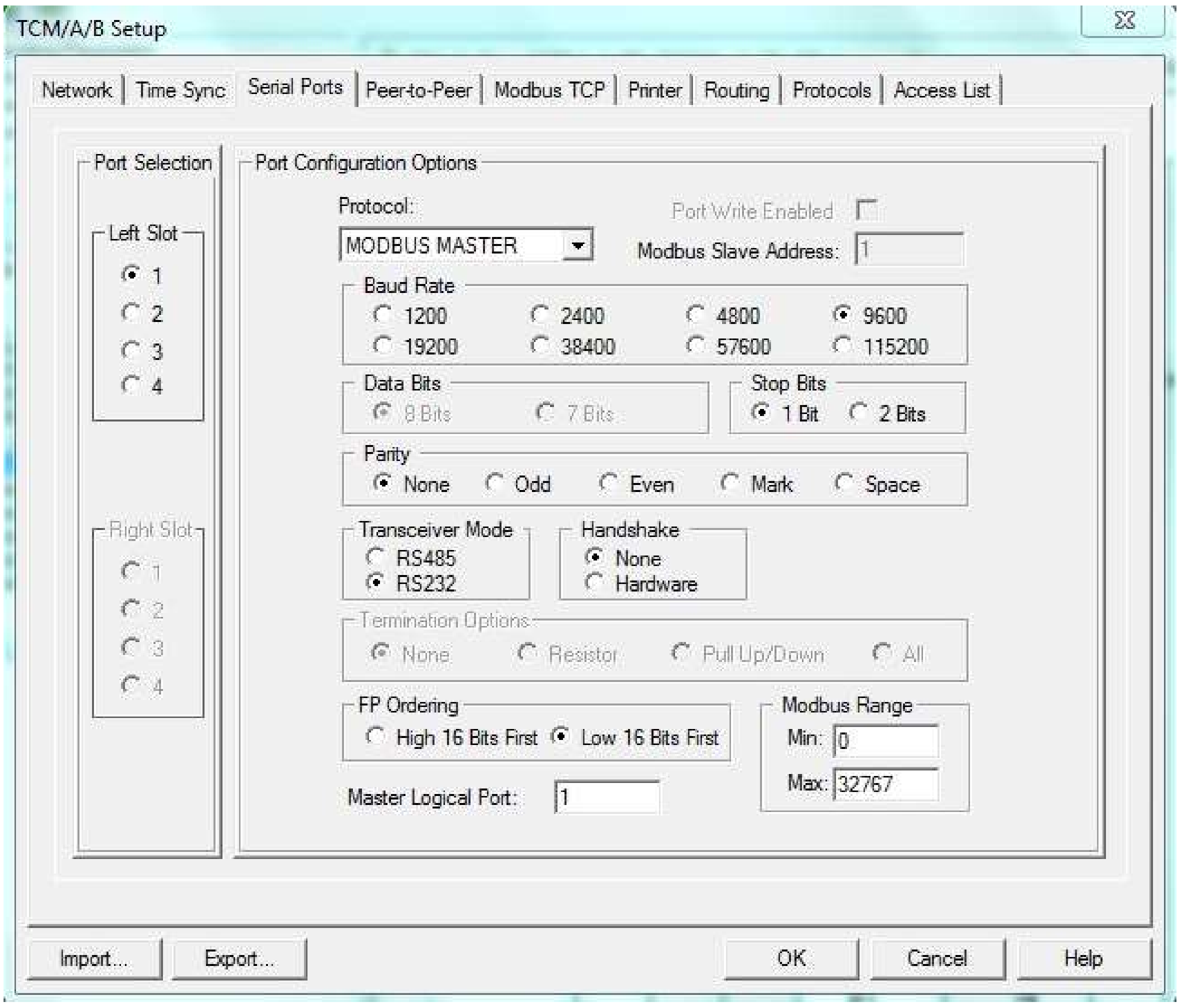
Using a Tricon Communication Module (TCM), the Tricon controller can communicate programming computers, other Tricon controllers, Ethernet devices and Modbus master and slave devices. A Tricon with a TCM can operate as the Modbus master or slave using the serial ports or by Modbus TCP using the NET1 or NET2 Ethernet ports.

A Tricon TCM serial port can act as the master, slave using the following physical connection features

- Point to point or multi drop network
- RS-232 or RS-422 or RS-485 communication standard
- Half duplex (2 wire) or full duplex (4 wire) cable wiring

Parameters for setting up the serial port include

- Port selection
- Port Write Enabled
- Protocol
- Modbus Slave Address
- Baud Rate
- Data Bits
- Stop Bits
- Parity
- Transceiver Mode
- Handshake
- Termination Options
- Floating Point Ordering
- Modbus Range
- Master Logical Port


TCM/A/B Setup

Network | Time Sync | Serial Ports | Peer-to-Peer | Modbus TCP | Printer | Routing | Protocols | Access List

Port Selection

Left Slot: [✓ 1] 2 3 4

Right Slot: 1 2 3 4

Port Configuration Options

Protocol: MODBUS MASTER

Port Write Enabled: ☐

Modbus Slave Address: 1

Baud Rate: 1200 2400 4800 [✓ 9600] 19200 38400 57600 115200

Data Bits: [✓ 8 Bits] 7 Bits

Stop Bits: [✓ 1 Bit] 2 Bits

Parity: [✓ None] Odd Even Mark Space

Transceiver Mode: RS485 [✓ RS232]

Handshake: [✓ None] Hardware

Termination Options: [✓ None] Resistor Pull Up/Down All

FP Ordering: High 16 Bits First [✓ Low 16 Bits First]

Modbus Range: Min: 0 Max: 32767

Master Logical Port: 1

Import... Export... OK Cancel Help

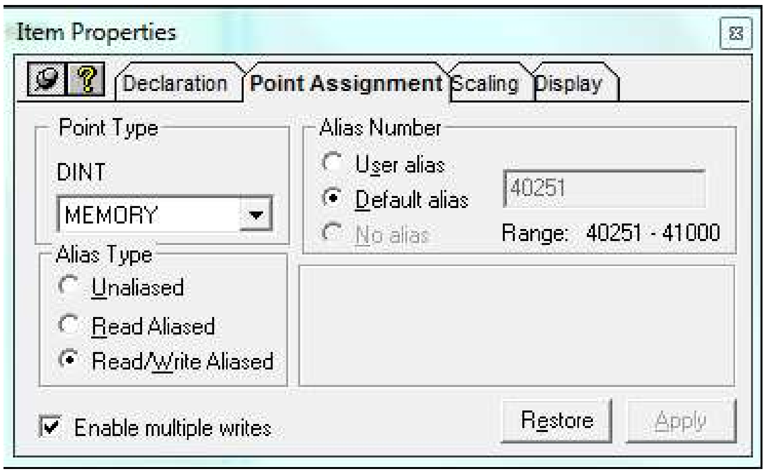
Figure 3.2 Example Screen TCM Serial Port Configuration

If using Modbus TCP, the following parameters are required for setup

- Protocol
- Port Write Enabled
- Master Logical Port
- TCP Port
- Network Port
- IP address
- Floating Point Ordering
- Modbus Range

3.2. Triconex Slave

By setting the serial port on a Tricon controller as a Modbus Slave, an external Master can read tags that have been assigned Modbus addressing via an alias as displayed in Figure 3.3.



Point Type
DINT
MEMORY

Alias Type
( ) Unaliased
( ) Read Aliased
(●) Read/Write Aliased

Alias Number
( ) User alias
(●) Default alias
( ) No alias
40251
Range: 40251 - 41000

[✓] Enable multiple writes

Figure 3.3 Tag aliased to Modbus addressing

Once this setup is complete, the Tricon is ready to serve as a Modbus Slave. When configuring the Tricon it must be noted that the Triconex supports Boolean, 32 bit double integers and 32 bit real numbers where the Modbus protocol supports only Booleans and 16 bit integers. This means by default for a double integer only the least significant 16 bits are transmitted and for a real number, the number must be scaled to a 16 bit integer or mapped to to 16 bit integers where one is whole number and the other is the decimal component.

3.3. Triconex Master

When configured as a Modbus master, the Tricon needs application code as well as the serial or Ethernet port configuration. The application logic uses function blocks to execute the function code commands. The read and write function blocks are available in the Tricon library

- MBREAD_BOOL
- MBREAD_DINT
- MBREAD_REAL
- MBREAD_REAL_TRD
- MBWRITE_BOOL
- MBWRITE_DINT
- MBWRITE_REAL
- MBWRITE_REAL_TRD

The function blocks that of the data type BOOL and DINT can each transmit up to 32 data values. The function blocks of the type REAL are limited to 25 data values.

For both the read and write function blocks, there are parameters that the function need to be configured to enable successful operation being

- Starting alias which is the first Modbus register in the slave device
- Number of registers to read or write
- Tricon TCM serial port number that communication will occur on
- Slave ID number of the slave device

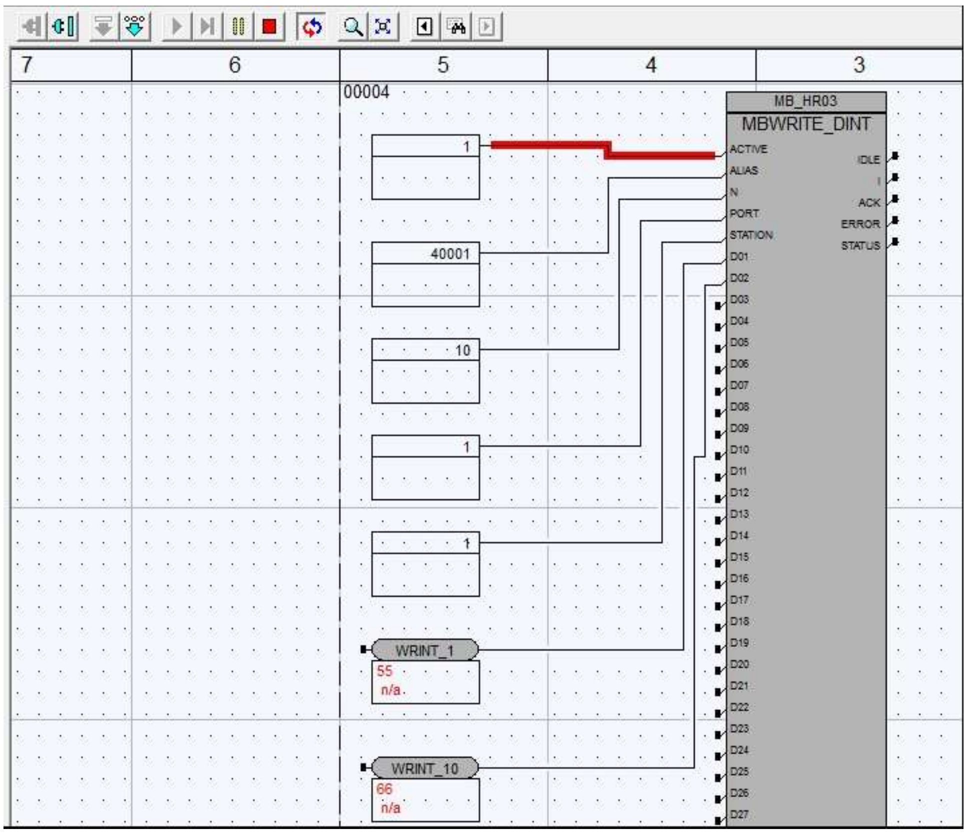


Figure 3.4 Tricon function block MBWRITE_DINT

3.4. Allen Bradley SLC500

An Allen Bradley SLC500 is Programmable Logic Controller (PLC) from the small control system suite of products from Rockwell Automation. The SLC500 has the following features

- Ladder logic and structured text programming
- Advanced instruction set
- Built in RS-232 and RS-422 communication port on the processor
- DeviceNet and ControlNet communications on selected processors
- On-board ports available for Data Highway Plus (DH+) or universal remote I/O
- Ethernet/IP communications available on selected processors

Figure 3.5 Allen Bradley SLC500 (Aotewell 2016)

The SLC500 does not have Modbus communications capability with any processor options, but with the addition of a Prosoft communications card such as the 3150-MCM, the processor is able to read and write across the backplane to the 3150-MCM module. The 3150-MCM then communicates to the connected Modbus device. The 3150-MCM can be setup as either a master or a slave device. There are two serial ports on the 3150-MCM and they can be configured for either RS-232 or RS-485.

Figure 3.6 Prosoft 3150-MCM (Prosoft 2014)

There are two serial ports on the 3150-MCM and they can be configured for either RS-232 or RS-485. Figure 3.7 displays the wiring requirements for a RS-232 configuration and Figure 3.8 shows the wiring for a RS-485 configuration.

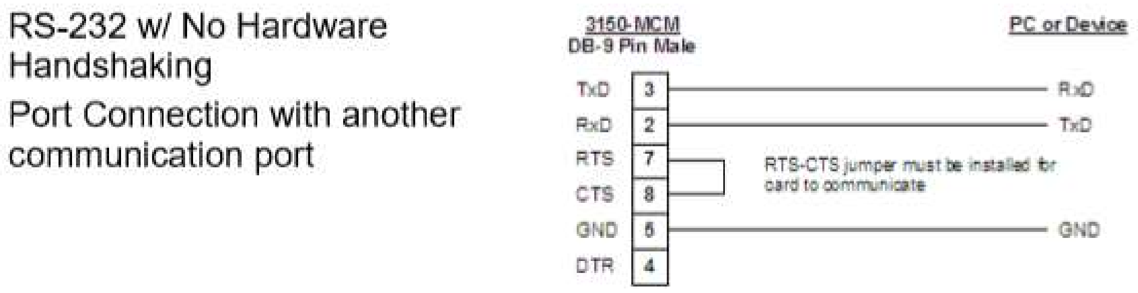

RS-232 w/ No Hardware Handshaking
Port Connection with another communication port
3150-MCM
DB-9 Pin Male
PC or Device
TxD 3 RxD
RxD 2 TxD
RTS 7
CTS 8
RTS-CTS jumper must be installed for card to communicate
GND 5 GND
DTR 4

Figure 3.7 3150-MCM RS-232 (Prosoft 2014)

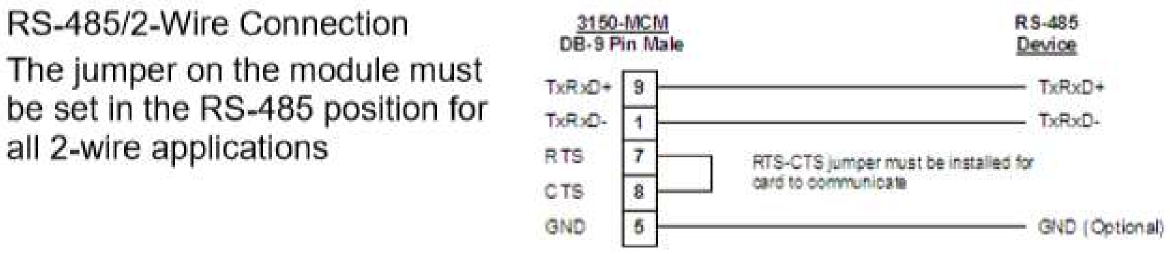

RS-485/2-Wire Connection
The jumper on the module must be set in the RS-485 position for all 2-wire applications
3150-MCM
DB-9 Pin Male
RS-485 Device
TxRxD+ 9 TxRxD+
TxRxD- 1 TxRxD-
RTS 7
CTS 8
RTS-CTS jumper must be installed for card to communicate
GND 5 GND (Optional)

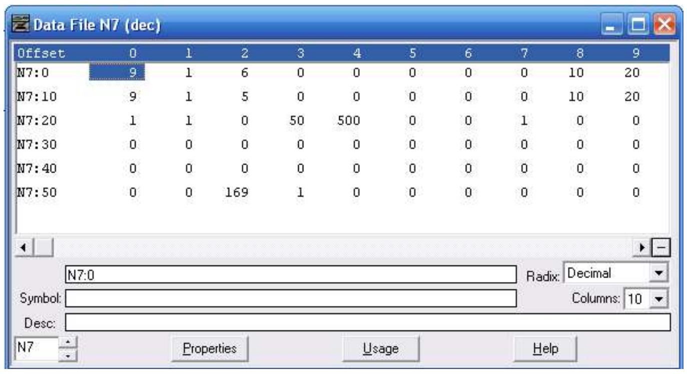
Figure 3.8 3150-MCM RS-485 (Prosoft 2014)

3.5. SLC500 Slave

The SLC500 is a register based control system. As such, the configuration for setting up the communications port and the Modbus definitions is done by setting registers in data table.

For example, the following configuration is used to setup the SLC500 as a Modbus Slave:

- N7:0 Bits 2 1 0. Set 000 to enable as RTU Slave
- N7:0 Bit 3. Set to 1 to allow Pass through enabled
- N7:0 Bit 4. Set to 0 to disable routing
- N7:0 Bits 13 12. Set to 00 for one stop bit
- N7:0: Bits 15 14. Set to 00 for no parity
- N7:1. Set integer value to 1 to set Slave address as 1
- N7:2. Set integer value to 5 to set baud rate to 9600
- N7:3. Set integer value to 0 to set RTS to TXD delay of 0
- N7:4. Set integer value to 0 to set RTS off delay to 0
- N7:5. Set integer value to 0 to set message response timeout to 0
- N7:7. Set integer to 0 to allow for Modbus register 10001 or 30001 to begin at 0
- N7:8. Set integer value at 10 to allow offset for function codes 01, 05, 15. To locate output image at word 100 enter 100 instead of 10 as an example. A function code 1 command with an address of zero will then start reading at 100
- N7:9. Set integer to 100. Function codes 03, 06, 16 use this a reference for starting point. To start 40001 at 100, enter 100



Data File N7 (dec)

Offset	0	1	2	3	4	5	6	7	8	9
N7:0	9	1	6	0	0	0	0	0	10	20
N7:10	9	1	5	0	0	0	0	0	10	20
N7:20	1	1	0	50	500	0	0	1	0	0
N7:30	0	0	0	0	0	0	0	0	0	0
N7:40	0	0	0	0	0	0	0	0	0	0
N7:50	0	0	169	1	0	0	0	0	0	0

N7:0 Radix: Decimal
Symbol: Columns: 10
Desc:
N7

Figure 3.9: SLC500 N7 file for MCM configuration

3.6. Signal Interfaces

As they are commonly used in Modbus communication links, a review and test of interface conversion hardware has been completed. As RS-485 is more tolerant to noise than RS-232, it is used on longer runs. For Modbus Master and Slave devices which only have RS-232 serial ports, RS-232 to RS-485 converters are used to change the physical connection. The Advantech Adam-4520 displayed in Figure 3.10 is one such device. It has a RS-232 DE9 connection, with RS-485 as a screw terminal connection. Baud rate is set by a series of dip switches located within the unit.

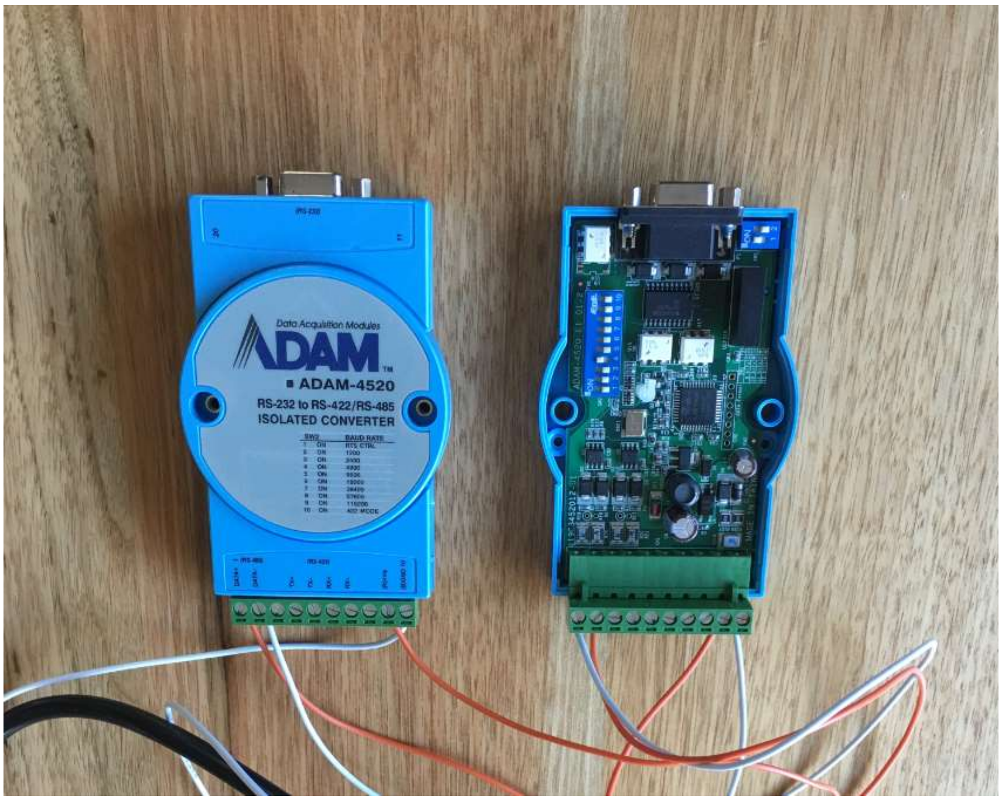


Figure 3.10: Adam-4520 RS-232 to RS-485 converter

The RS-232 to RS-485 converters can also be used when using a standard laptop or personal computer as a simulation tool. Onboard serial ports are mostly RS-232 unless they are a special configuration. However, it is common for modern computers to not have a serial port so an option is to use a terminal server that converts Ethernet to RS-232 or RS-485. The Lantronix Xpress DR serial to Ethernet terminal is one such device. The Xpress DR is compact as displayed in Figure 3.11 but requires an external 24volt direct current power source.

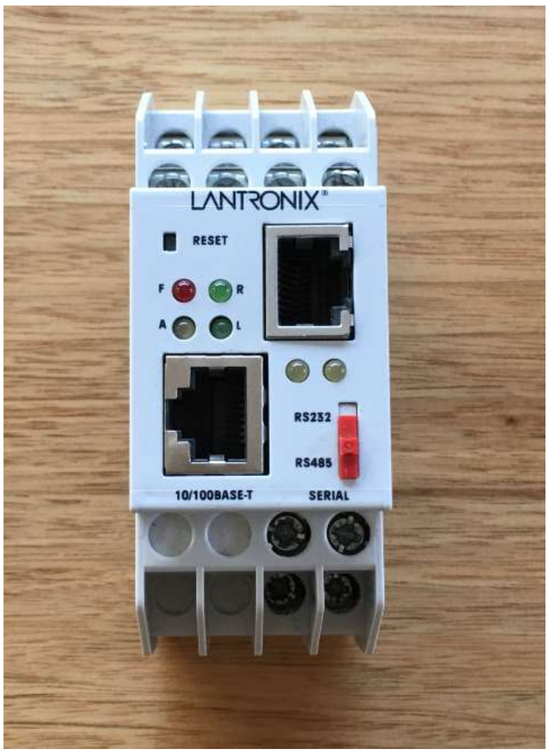


Figure 3.11: Lantronix Xpress DR terminal server

The most common option is to convert a RS-232 computer interface to RS-485 is to use a simple converter such as the unit in Figure 3.12. It is self-powered, using the transmit and receive signals as the power supply. A DB9 female connector is for the RS-232 to computer connection, and a 4 terminal block for the RS-485 side. It has a transmission distance of 1200m for RS-485 and 5m for RS-232.

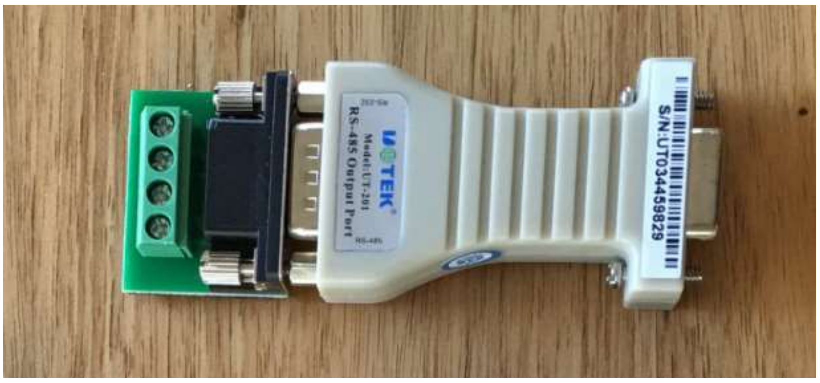


Figure 3.12: UTEK UT201 RS-232 to RS-485 converter

A passive device used to assist in the physical network configuration was the Weidmuller DB9 to screw terminal (Figure 3.13). This din rail mountable unit was useful by using a DB9 serial straight through cable to connect to the Triconex and SLC500 serial ports, the screw terminals are then easily accessible allowing for easy serial communications configuration. For example, RS-232 on the SLC500 uses pins 2,3 and 5 with a jumper required between 7 and 8. RS-485 on the SLC500 uses pins, 9,1 & 5 with a jumper between 7 and 8. Without the DB9 screw terminal block, multiple DB9 serial cables would need to be configured. This option allows for one cable with a DB9 at one and open ended cable with the individual wires boot laced pinned which allows for flexibility in changing physical connections.

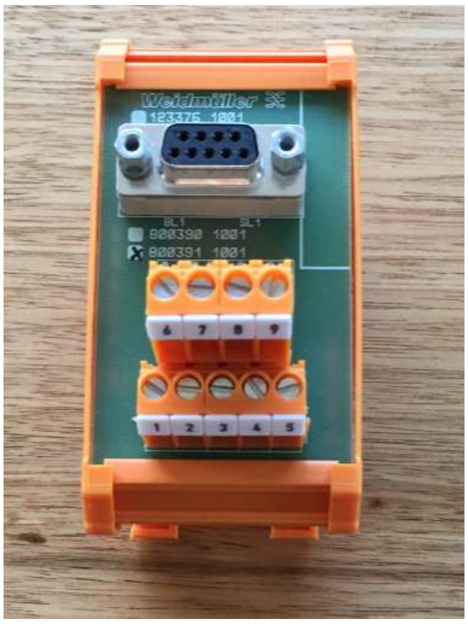


Figure 3.13: Weidmuller DB9 Screw Terminal

3.7. Test Systems

Before developing the Arduino Modbus Simulator hardware and software solution, test systems using common control systems have been implemented. The justification for doing so was to build confidence in having a working system before substituting in Arduino based prototypes. It also provides the ability to study how Modbus is implemented in different vendor software.

Four different systems were built being

- ModScan Master to Triconex Slave
- ModScan Master to SLC500 Slave
- Triconex Master to ModSim Slave
- Triconex Master to SLC500 Slave

A Modbus Master to Modbus Slave network was built using WinTech ModScan as the Master and an Invensys Triconex as the Slave (Figure 3.13). The message encoding format used in the system was RTU. The network was built and tested using RS-232 and then RS-485. When using RS-485, the UTEK UT201 RS-232 to RS-485 converter was used at the computer end of the network, while for the Triconex the same DB9 serial port is used but software selection in the Triconex enables selection between RS-232 and RS-485. Another solution for testing communication over RS-485 was completed by using the Lantronix Xpress terminal server. The Lantronix was connected to the Ethernet port of the computer running ModScan, with the terminal server being configured for Modbus RTU RS-485 connection. The Lantronix was physically wired RS-485 to the Triconex serial port which was software configured for RS-485. A test configuration for the Triconex was built using 10 holding registers and 10 coils. The Master was able to read and write to these registers.

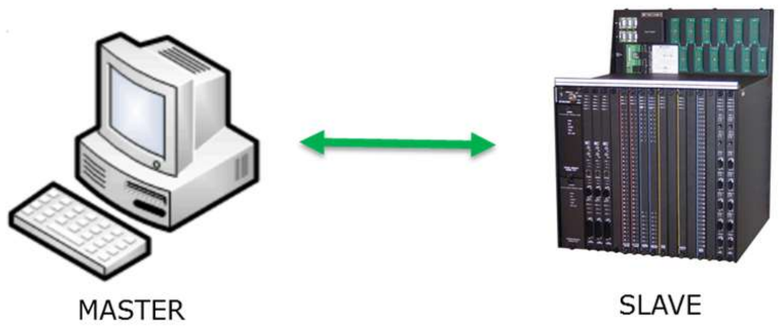


Figure 3.14: ModScan to Triconex

A second Modbus Master to Modbus Slave network was built using an Invensys Triconex as the Master and WinTech ModSim as the Slave (Figure 3.14). The message encoding format used in the system was RTU. The network was built and tested using RS-232 and then RS-485. When using RS-485, the UTEK UT201 RS-232 to RS-485 converter was used at the computer end of the network, while for the Triconex the same DB9 serial port is used but software selection in the Triconex enables selection between RS-232 and RS-485. A test configuration for the Triconex was built using the following function blocks

- MBREAD_BOOL
- MBREAD_DINT
- MBWRITE_BOOL
- MBWRITE_DINT

The Boolean logic was setup to read 5 coils and write 5 coils. The register function blocks were setup to read 5 registers and write 5 registers.

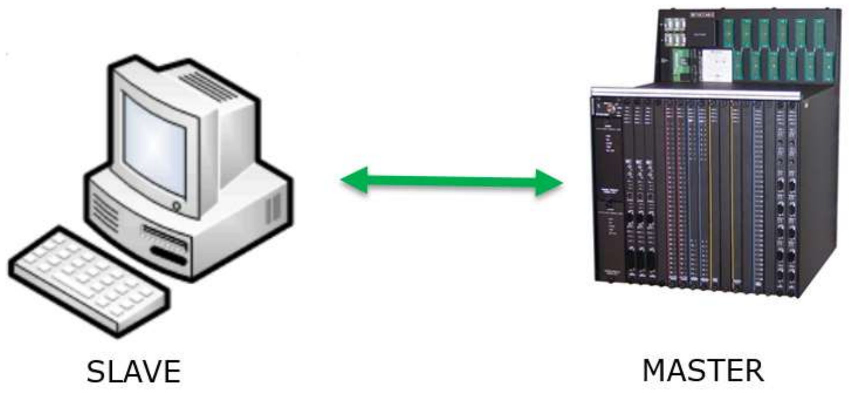


Figure 3.14: Triconex to ModSim

A third Modbus Master to Modbus Slave network was built using WinTech ModScan as the Master and an Allen Bradley SLC500 as the Slave (Figure 3.15). The message encoding format used in the system was RTU. The network was built and tested using RS-232. A test configuration for the SLC500 was built using Modbus registers for function codes 1 to 4. As the SLC500 is a data table based controller, the Modbus address was mapped to the following N7 data table values

- Write to Coils SLC500=N10:9 bits 0 to 9 ModScan=00001 to 00010
- Read from Digital Inputs SLC500=N10:0 bits 0 to 9 ModScan=10001 to 10010
- Write to Holding Registers SLC500=N10:20 to N10:29 ModScan=40001 to 40010
- Read from Analogue Inputs SLC500=N10:10 to N10:19 ModScan=30001 to 30010

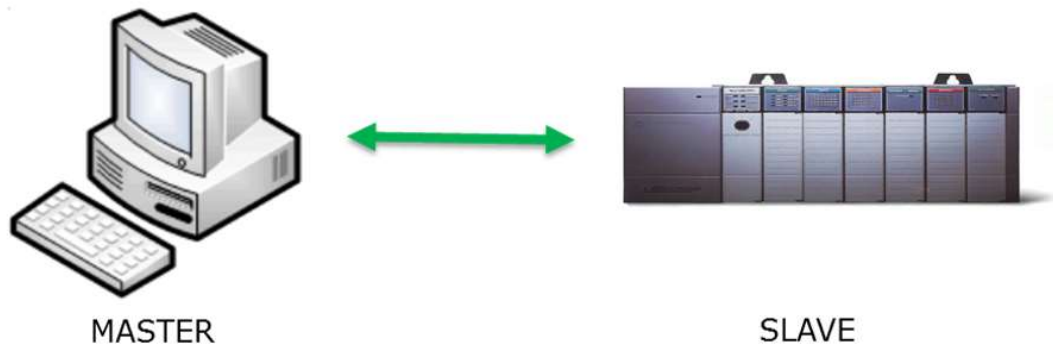


Figure 3.15: ModScan to SLC500

A fourth Modbus Master to Modbus Slave network was built using an Invensys Triconex as the Master and an Allen Bradley SLC500 as the Slave (Figure 3.14). The message encoding format used in the system was RTU. The network was built and tested using RS-232 at the Triconex and SLC500 but converted to RS-485 using the Adam 4520 RS-232 to RS-485 converters. This proved a RS-485 network between the two controllers which is a common industrial communications network. A test configuration for the Triconex was built using the following function blocks

- MBREAD_BOOL
- MBREAD_DINT
- MBWRITE_BOOL
- MBWRITE_DINT

The Boolean logic was setup to read 5 coils and write 5 coils. The register function blocks were setup to read 5 registers and write 5 registers. The SLC500 data table was configured with the following register setup

- Write to Coils SLC500=N10:9 bits 0 to 4 mapped Modbus registers=00001 to 00005
- Read from Digital Inputs SLC500=N10:0 bits 0 to 4 mapped Modbus registers=10001 to 10005
- Write to Holding Registers SLC500=N10:20 to N10:24 mapped Modbus register =40001 to 40005
- Read from Analogue Inputs SLC500=N10:10 to N10:04 mapped Modbus register=30011 to 30015

All 4 networks were successfully built with communications between devices were established.

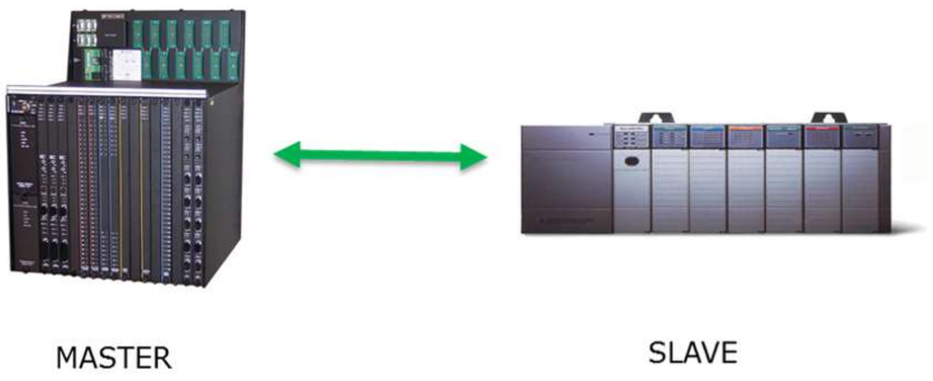


Figure 3.16: Triconex to SLC500

4. Hardware Analysis and Design

This section of the dissertation focuses on a review of the hardware options available and the strengths and weaknesses of each component relative to achieving the project goal. While the main goal of producing an Arduino Modbus Simulator is a Master, using RS-485 as the transport layer and using RTU as the message encoding format, if in achieving this minimum requirement other options are available to extend capability with zero impact to the project they were also evaluated.

The hardware required to complete the requirements can be broken in the following categories

- Microprocessor
- RS-485 serial communications
- RS-232 serial communications
- Display
- Input keypads
- Enclosure
- Ancillaries

Each component is discussed and assessed again a ranking for that component type before the final choice is documented.

4.1. Microprocessor Analysis

As part of the hardware evaluation three Arduino or Arduino compatible microcontrollers were considered during the project. As displayed in Figure 4.1 the three microcontrollers are

- Arduino MEGA 2650 (Figure 4.1 Left)
- Arduino UNO (Figure 4.1 Centre)
- Teensy 3.2 (Figure 4.1 Right)

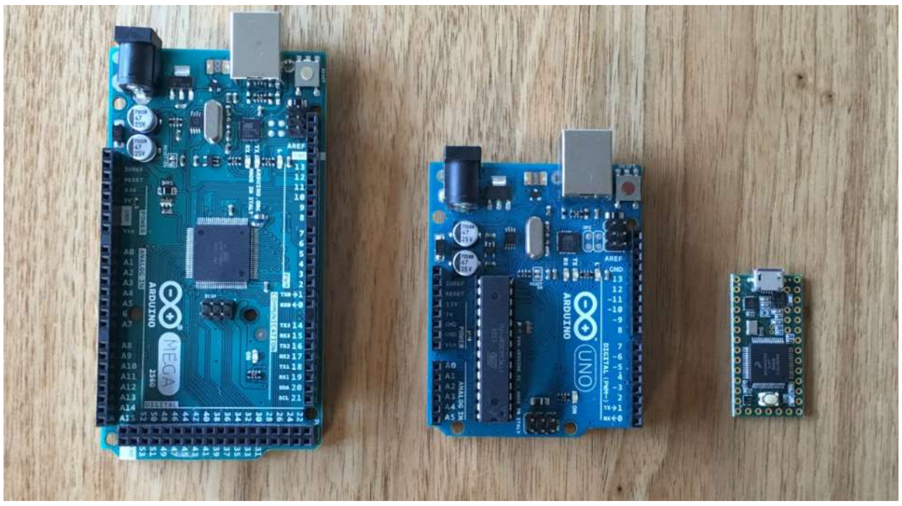


Figure 4.1: Microcontroller options

Initially an Arduino Mega 2650 was the microcontroller board used during the first prototyping. The MEGA has good memory and a lot of input / output pins options and 3 UARTs for serial communications. The downside of the MEGA option is that is has a large footprint.

The second microcontroller evaluated was an Arduino UNO. The UNO processor runs at the same speed, 16MHz, but has less memory and less input / output capacity. It is also limited to one UART. It is compact in size in comparison to the MEGA.

The third and last microcontroller tested was the Teensy 3.2. Teesny is not an Arduino product but is programmed using the Arduino IDE. The Teensy has an extremely fast processor at 96MHz, large memory, and an input / output count in between the UNO and MEGA count. The primary advantage of the Teensy is it is extremely compact with a very small footprint. The small disadvantage that the Teensy has in comparison to the MEGA and UNO is that it requires on board soldering of the connections to the input and outputs. The MEGA and UNO have the alibility for wiring to be soldered to pins before insertion into the terminals. The risk this brings to the Teensy is excessive heat on the microcontroller board leading to some form of damage or cross connection between input and outputs due to poor soldering,

A summary of the specifications of each microcontroller is provided in Table 4.1

Microcontroller	**CPU Speed**	**Memory**	**Input / Output / Serial**
Arduino UNO	16 MHz	8KB SRAM 4KB EEPROM	54 DIO 16 AI 3 Serial
Arduino MEGA	16 MHz	2KB SRAM 1KB EEPROM	14 DIO 6 AI 1 Serial
Teensy 3.2	96 MHz	64KB SRAM 2KB EEPROM	34 DIO 21 AI 3 Serial

Table 4.1: Microcontroller Specifications

Using a ranking system of one being the best and three being the least suitable, each microcontroller was ranked again four criteria being

- CPU speed
- Memory
- Input / Output / Serial capacity
- Physical size

The results of the ranking are displayed in Table 4.2. From the perspective of pure hardware and not software, the Teensy 3.2 is the most suitable, followed by the MEGA with the UNO last. However, due to issues with software compatibility with library files, the Teensy was eliminated from consideration for the prototype.

Microcontroller	**CPU Speed**	**Memory**	**Input / Output / Serial**	**Physical Size**
Arduino UNO	2	3	3	2
Arduino MEGA	2	2	1	3
Teensy 3.2	1	1	2	1

Table 4.2: Microcontroller rankings

4.2. RS-485 Serial Communication Analysis

As part of the hardware evaluation five RS-485 serial communication hardware options were considered during the project. As displayed in Figure 4.2 the five options are

- MAXUM RS-485 Integrated Circuit (Figure 4.2 top left)
- Altronics TTL to RS-485 board (Figure 4.2 top centre)
- Libelium RS-485 shield (Figure 4.2 top right)
- DFRobot RS-485 shield (Figure 4.2 bottom left)
- Linksprite RS-485 shield (Figure 4.2 bottom right)

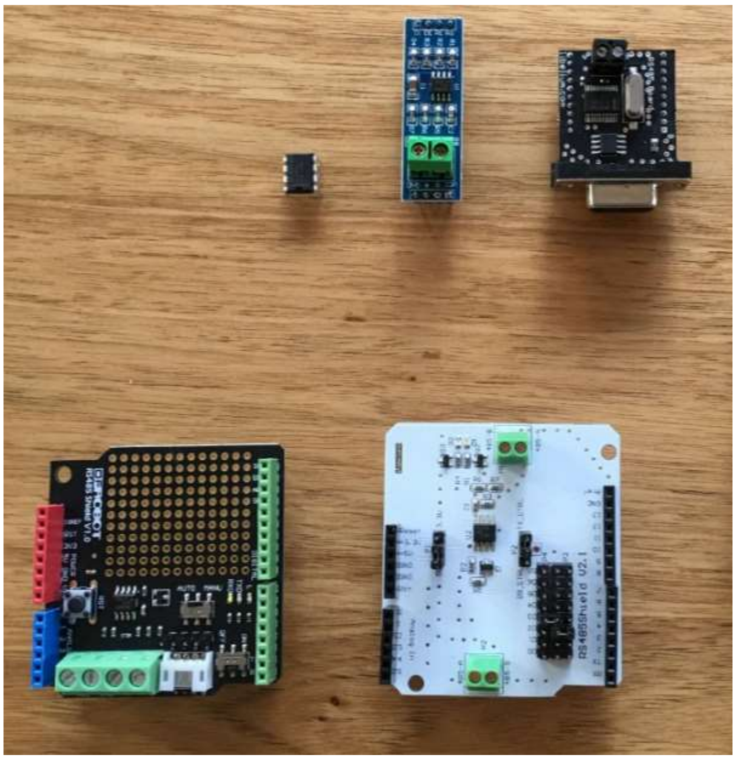


Figure 4.2: RS-485 serial hardware

The MAXIM-485 integrated circuit offers both flexibility and appears to offer low footprint. However, the IC requires supporting electronic circuitry which when combined onto a small printed circuit board, grows the footprint.

The Altronics TTL to RS-485 module is slender, has all the required supporting circuitry and has screw terminal connection for the end device connection. This breakout module requires an enable pin to be pulled high to allow for serial communication to occur. While this is not an absolute requirement for RS-485 communication, this board requires that wiring modification to be completed if allowing serial communications to be automatic.

The Libelium RS-485 shield is compact, has optional screw terminal or DB9 for the RS-485 connection, and includes all supporting circuitry. However, it requires a mother shield that this is placed onto the Arduino, making the footprint four times the size.

The DFRobot RS-485 shield is not compact, only allows for the serial connection to be from one pair of designated pins and is designed to sit on top of an Arduino host. It does have all the supporting circuitry including a selector switch that determines if an enable pin is to be used or automatically pulling it high to allow communication at all times.

The LinkSprite RS-485 shield is not compact and is designed to sit on top of an Arduino board. It has all the supporting circuitry to enable serial communication and its biggest advantage is it allows for a range of outputs to be used as the serial communications. By using jumpers, outputs zero to seven can be selected giving flexibility. This is valuable if using an Arduino Uno and allows the fixed UART serial port (outputs zero to one) to be used for other purposes and by using a software serial port, it can be mapped the any of the other outputs on the LinkSprite.

The initial testing used for evaluating these boards was to use two Arduino Uno's, each with a RS-485 serial communication device, and setup one as the sender and one as the receiver. Using the SoftwareSerial function from the Arduino IDE, a simple ASCII message was sent from the sending device to the receving device and the serial monitor function included in the Arduino IDE was used to display the value on a computer connect to the receiving Arduino. All hardware options were successful.

The ranking of the RS-485 hardware is given in table 4.3. A ranking of one indicates the most suitable and a higher number is less suitable. If hardware is equal and cannot be differentiated, then an equal ranking is given.

Board	**Size**	**Circuitry Complete**	**Auto Enabled**	**Flexible Connection**
Maxim	3	5	2	1
Altronics	1	2	2	1
Libelium	5	3	2	3
DFRobot	2	1	1	2
LinkSprite	4	1	1	1

Table 4.3: RS-485 rankings

The results of the ranking show that the Altronics TTL to RS-485 board and the DFRobot are the best hardware options,

4.3. RS-232 Serial Communication Analysis

As part of the hardware evaluation five RS-232 serial communication hardware options were considered during the project. As displayed in Figure 4.3 the five options are

- MAXUM RS-232 Integrated Circuit (Figure 4.3 top right)
- Altronics TTL to RS-232 board (Figure 4.3 top left)
- Libelium RS-232 shield (Figure 4.3 bottom right)
- DFRobot RS-485 shield (Figure 4.3 bottom centre)
- Linksprite RS-485 shield (Figure 4.3 bottom left)

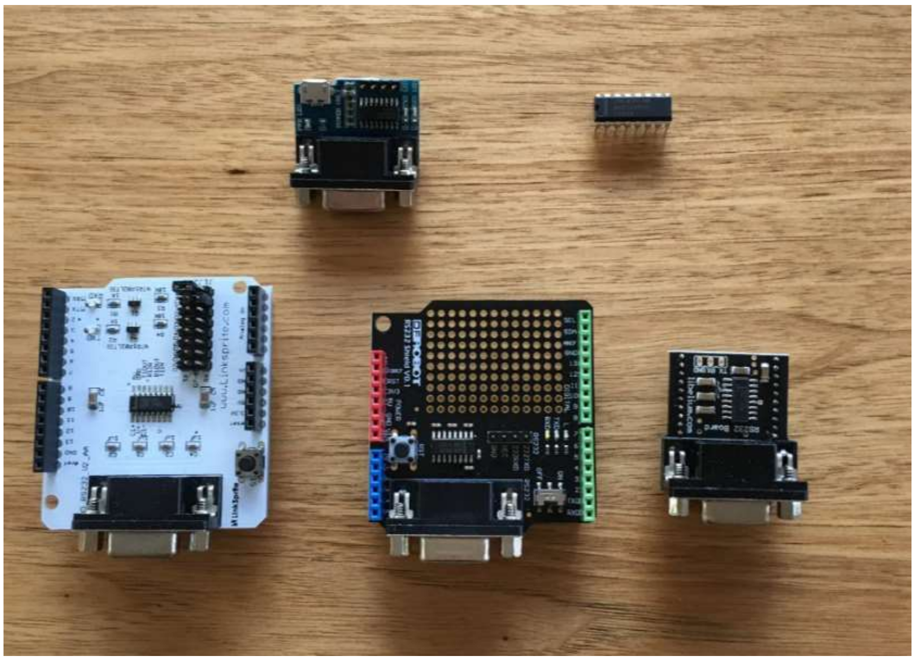


Figure 4.3: RS-232 serial hardware

The MAXIM-232 integrated circuit offers both flexibility and appears to offer low footprint. However, the IC requires supporting electronic circuitry which when combined onto a small printed circuit board, grows the footprint.

The Altronics TTL to RS-232 module is compact, has all the required supporting circuitry and has a DB9 for the end device connection.

The Libelium RS-232 shield is compact, has a DB9 for the RS-232 connection, and includes all supporting circuitry. However, it requires a mother shield that this is placed onto the Arduino, making the footprint four times the size.

The DFRobot RS-232 shield is not compact, only allows for the serial connection to be from one pair of designated pins and is designed to sit on top of an Arduino host. It does have all the supporting circuitry.

The LinkSprite RS-232 shield is not compact and is designed to sit on top of an Arduino board. It has all the supporting circuitry to enable serial communication and its biggest advantage is it allows for a range of outputs to be used as the serial communications. By using jumpers, outputs zero to seven can be selected giving flexibility. This is valuable if using an Arduino Uno and allows the fixed UART serial port (outputs zero to one) to be used for other purposes and by using a software serial port, it can be mapped the any of the other outputs on the LinkSprite.

The initial testing used for evaluating these boards was to use two Arduino Uno's, each with a RS-232 serial communication device, and setup one as the sender and one as the receiver. Using the SoftwareSerial function from the Arduino IDE, a simple ASCII message was sent from the sending device to the receiving device and the serial monitor function included in the Arduino IDE was used to display the value on a computer connect to the receiving Arduino. All hardware options were successful.

The ranking of the RS-232 hardware is given in table 4.4. A ranking of one indicates the most suitable and a higher number is less suitable. If hardware is equal and cannot be differentiated, then an equal ranking is given.

Board	**Size**	**Circuitry Complete**
Maxim	3	5
Altronics	1	2
Libelium	5	3
DFRobot	2	1
LinkSprite	4	1

Table 4.4: RS-232 rankings

The results of the ranking show again that the Altronics TTL to RS-232 board and the DFRobot are the best hardware options.

4.4. Display

As part of the hardware evaluation three LCD display options were considered during the project. As displayed in Figure 4.4 the three displays are

- DFR0091 128x64 LCD (Figure 4.4 top left)
- FIT0328 128x64 LCD (Figure 4.4 top right)
- Standard 16x2 LCD (Figure 4.4 bottom)

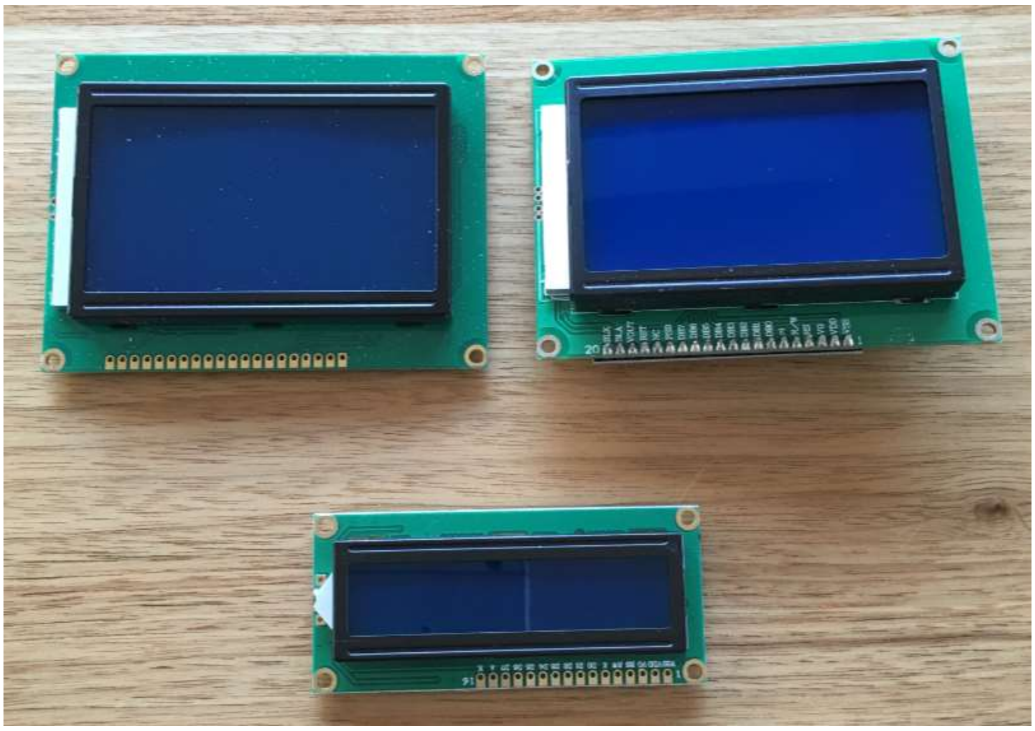


Figure 4.4: Display Options

The evaluation of the display options was a relatively simple process. The differentiator between the two 128x64 LCD options was the ability of the DFR0091 model to be connected to the Arduino by parallel mode (multiple wiring) or three wires serial. The smaller LCD 16x2 display offers parallel mode only.

4.5. Input Keypad

Two different membrane keypad options were considered as part of the hardware select. As depicted in Figure 4.5, a 4x3 matrix keypad and a 4x4 matrix keypad.

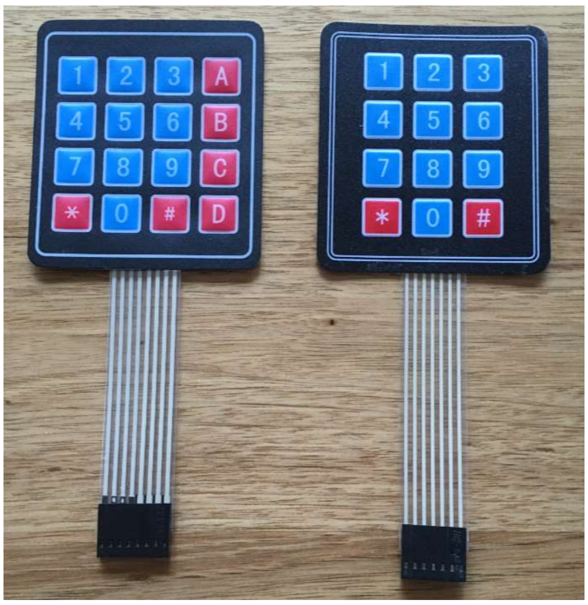


Figure 4.5: Keypad options

The 4x4 keypad is only marginally bigger in footprint than the 4x3 option but provides an extra four buttons. These buttons offer a simple but effective menu option.

4.6. Enclosure

Three different hardware enclosures were considered as part of the design. A trade off between size, ergonomic feel, weight and ability to mount internally the required electronics were the factors considered. As displayed in Figure 4.6, three displays were considered being

- Altronics Large
- Altronics Medium
- PacTec PPT-4081

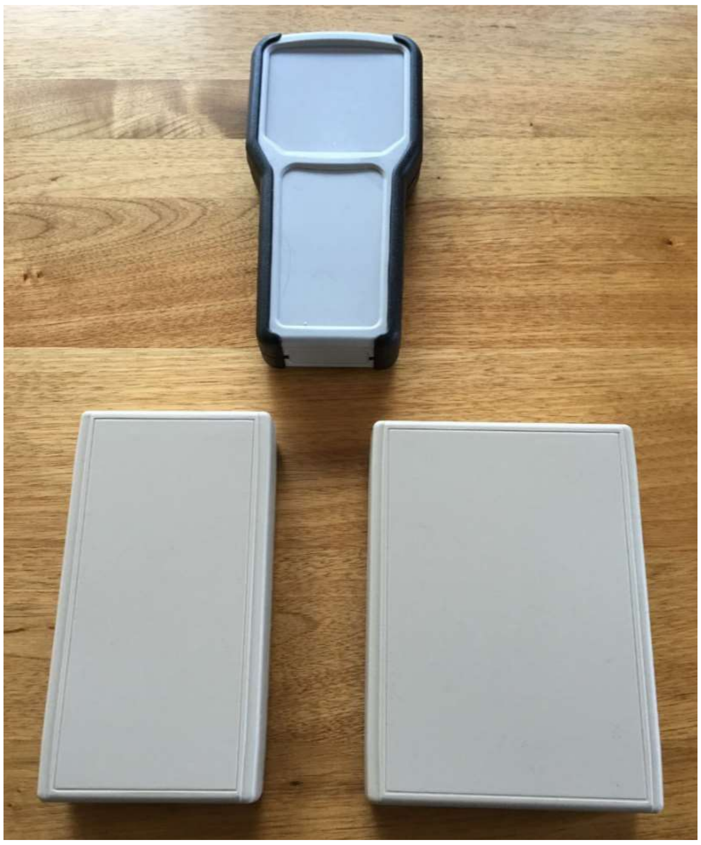


Figure 4.6: Enclosure Options

The first enclosure provided plenty of room internally allowing for the use of any Microcontroller and serial hardware options. The original prototype design was also using a 128x64 LCD display which was easily accommodated by this enclosure. It also had enough room to allow for two separate communication ports. This enclosure offered the most flexibility for the hardware design, but its most negative feature was ergonomics and footprint. It was not possible to easily hold with one hand, and didn't have a comfortable feel. The solid edges of the unit were also not aesthetically appealing. This enclosure was sourced from a local electronics outlet.

The second enclosure was a smaller version of the first. It traded off some of the size in the attempt to find a more ergonomic and aesthetically pleasing option. The smaller unit meant it was able to held in one hand quite easily, but came the trade of came with the enclosure unable to accommodate the 128x64 LCD display. Only a 16x2 LCD screen would fit.

The third enclosure option is PacTec PPT-4081. After the initial evaluation, a suitable enclosure meeting the ergonomic and aesthetic goals could not be sourced locally, A review of what enclosures are available in a open world market was conducted. Several options were considered before deciding to purchase and trial the PPT-4081. It was a four day purchase to delivery turnaround time from Pennsylvania, United States of America to Perth, Australia. This enclosure had the smallest internal size and only allowed for a 16x2 LCD screen. It did come with a battery enclosure with simple access door. The biggest advantage of this unit is the round edges complete with a soft rubber like finish. It comfortably can be held with one hand and it has a more professional visual finish compared to the other two enclosures evaluated.

Enclosure	**Screen Size**	**Internal Size**	**Ergonomic**	**Visual**
Altronics L	1	1	3	3
Altronics M	2	2	2	2
PPT-4081	2	3	1	1

Table 4.5: Enclosure rankings

After evaluation the PPT-4081has the best score. However, it essentially is trade off between ease of hardware design versus ergonomic feel. A factor to be considered here is that if a device is not easy to use, it will be relegated to the back of the test bench and will not be used.

4.7. Ancillaries

The other types of hardware that was considered and used during this project can be classed as ancillaries. This included serial ports, terminals, terminal strip, potentiometers, mini din connections, wiring, battery connectors, toggle switches and dip switches. When it came to the selection for the external apparatus, such as the switches and communications ports, being able to mount via a hole instead of an irregular shape was a definite advantage. By using mounting through a hole with a lock nut for mechanical securing, it is both faster and easier to simply drill a hole with a drill bit than try to cut and file a rectangular or other shape.

4.8. Prototype

Through the project, the Arduino Modbus Simulator prototype went through several prototyping stages. The different stages can be summarised as the following

- Arduino MEGA with RS-232
- Arduino MEGA with RS-485
- Arduino MEGA with 4x3 keypad
- Arduino MEGA with 16x2 LCD display
- Arduino MEGA with LCD display and keypad
- Arduino MEGA with selectable RS-232 and RS-485
- Arduino MEGA with LCD, keypad, RS-232 and RS-485
- Arduino MEGA with 4x4 keypad

In order to reduce footprint size, the Arduino MEGA 2650 was replaced with an Arduino Uno and the same prototype systems replicated.

The Teensy 3.2 was selected for evaluation when the project came to final design and difficulties in getting all the required hardware to fit within the selected enclosure. The Teensy prototyping was successful in a build including the LCD, keypad and both serial communication options, but an incompatibility issue with a Modbus Master library ruled it out from final system design.

4.9. Final Hardware Design & Assembly

At the completion of the hardware analysis, the final design included the following hardware

- Microcontroller - Arduino Uno
- Display - 16x2 LCD
- Input - 4x4 membrane keypad
- RS-485 Serial – DFRobot
- RS-232 Serial – Altronics TTL to RS-232
- Serial connection – 6 pin mini din
- Communications selector switch – toggle double pole
- Power switch – toggle double pole
- Potentiometer – A10K
- Battery – 9V D cell
- Enclosure – US Ergo

The circuit diagram is represented in Appendix B

The build process for the final design was approximately 10 hours. The process started by firstly marking the out the layout on the enclosure and then removing the unwanted material.

The cut out for the LCD was done by using a Dremel disc cutter and then a 2nd cut bastard file to smooth out. The same for the keypad membrane connection, The mini din connector, switches and potentiometer were simply drilled using the right diameter drill bit. The ease of mounting this equipment in comparison to the LCD reinforced why hole and lock nut fastening was a determining factor in selecting this hardware.

The enclosure also had some internal sockets used for mounting hardware. These were removed in order to maximise the available internal footprint but using a Dremel grinding disc to cutaway the moulded plastic plugs. Once the enclosure wad prepared the first of the hardware was mounted as represented by Figure 4.7.

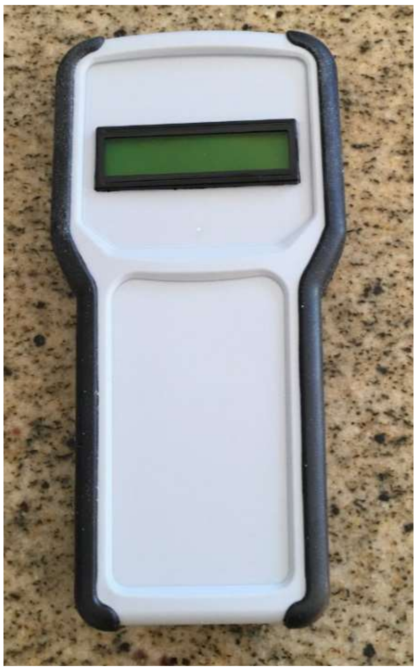


Figure 4.7: PPT-4081 enclosure with LCD

The non lock nut hardware such as the Arduino Uno and the communication modules have been mounted by using double sided tape to fix in position as displayed in Figure 4.8. Mechanical testing demonstrated the fixing was sturdy.

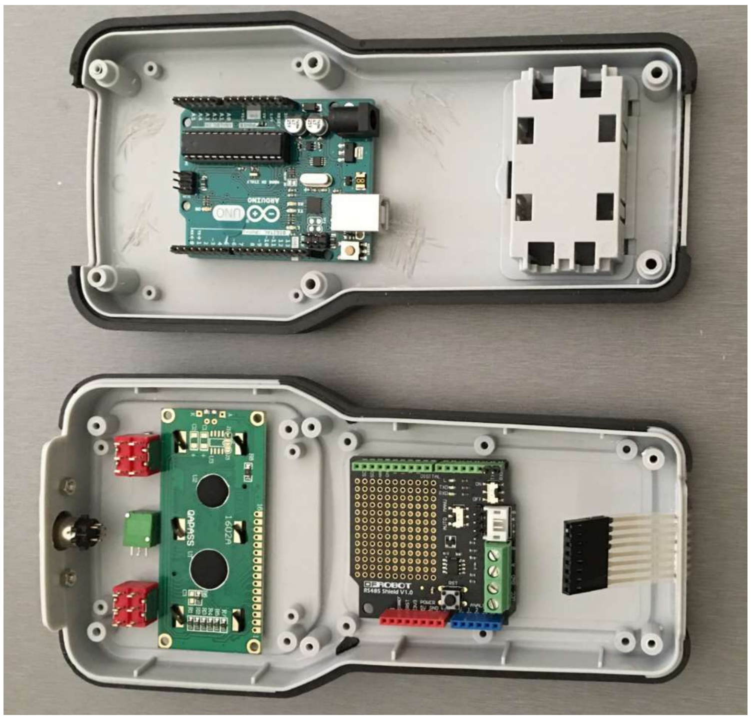


Figure 4.8: Mounting of hardware

Once the hardware was mounted in position, the wiring connections was made (Figure 4.9). The initial looming of wires was quite easy but as the number of wires grew, the difficulty increased.

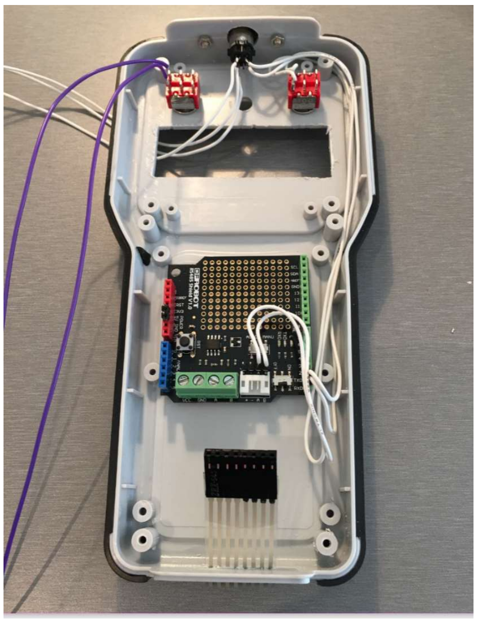


Figure 4.9: Initial wiring

Figure 4.10 shows the final wiring before it was loomed into position. Sticky squares were used along with cable ties for mounting the looms to the enclosure. Each wire also has a unique grafoplast number which makes initial wiring and trouble shooting easier.

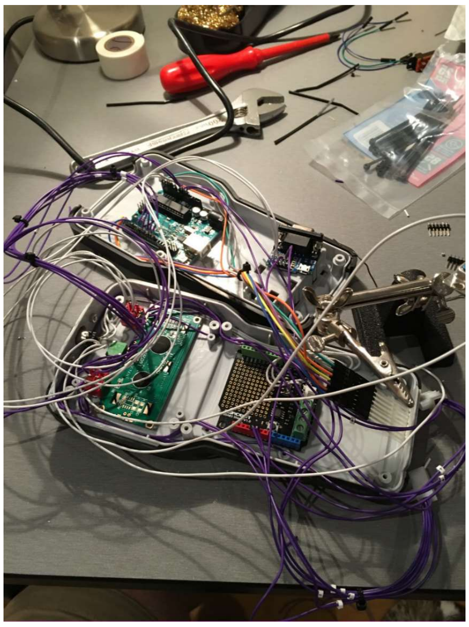


Figure 4.10: Wiring looms

Once the wiring was completed, the enclosure was assembled and the final outcome is displayed in Figure 4.11. The final hardware has met the hardware goal of being light, with weight of 400 grams. The unit is ergonomic and fits comfortably in one hand. The rounded edges have a soft feel and also make it aesthetical pleasing.

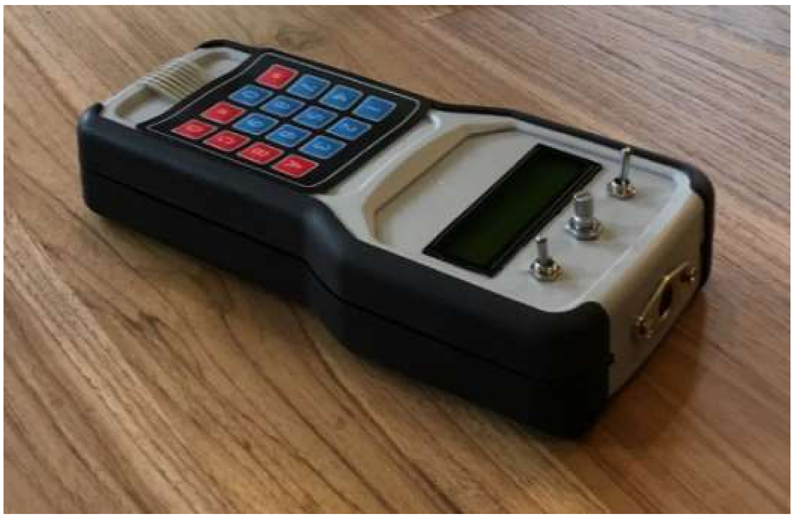


Figure 4.11: Final Hardware Design

5. Software Analysis and Design

This section of the dissertation focuses on the software development that was required in order to achieve the project goals. This chapter explains the Arduino IDE, the Modbus Master library selection, the Modbus Master software configuration, the Modbus Slave library selection and the Modbus Slave software configuration

5.1. Arduino Integrated Development Environment

The Arduino language is a simple language that creates a loop that runs constantly on the microcontroller. The mandatory syntax of a valid Arduino program is as follows

```
void setup () {

}

void loop () {

}
```

The setup() and loop() functions must always be present. The setup() function contains the code that is run only once by the microcontroller when the microcontroller is first powered up. The loop() function contains the code that loops forever until the power is reset or removed.

Specific libraries can be included by declaring them at the beginning program using the #include syntax.

Functions can also be added. These do not need to be declared before first using it, as they can be declared later in the program.

After all of the above is completed, the IDE passes the program to a C++ compiler to finish compilation before downloading to the microcontroller.

5.2. Arduino Modbus Simulator - Master

Two different software solutions have been developed using the hardware. The first acts as a Modbus Master and the second acts as a Modbus Slave. This section details the development of the Master software solution.

After conduction a review of the open source libraries available for a Modbus Master including basic testing of functionality, an open source library ModbusRtu has been selected. ModbusRtu is freely available via GitHub download. A key advantage of using this library is it allows the Modbus serial communication to be executed using the SoftwareSerial library. This library replicates the functionality of the UART, allowing any digital pins on the Arduino to be used for serial communication. The main benefit this offers is allowing the native serial communication port, pins 0 and 1 on an Arduino Uno, to be used with the serial monitor application that is built into the Arduino IDE. The native serial port is connected to the USB connection which allows the serial monitor program to be used for troubleshooting during the software development process.

The program for the Modbus Master has been written in the Arduino IDE. The program structure can be broken down into the following sections

- Include Libraries
- Declaration of Variables
- Setup Code (ran once upon power initialisation)
- Main Code (ran in a continuous loop)
- Keypad Operation (a function called when the keypad is activated

There are four libraries used in this program being

- ModbusRtu
- SoftwareSerial
- Keypad
- LiquidCrystal

ModbusRtu allows for communicating with Modbus devices using the serial interfaces, RS232, RS485 and USB, via the RTU message encoding protocol. It provides the query structure that allows the Master to generate the query using the correct fields matching the Modbus specification.

As previously discussed, the SoftwareSerial library allows for serial communication on any digital pins besides those directly connected to an onboard UART. This library is inherent to the Arduino IDE.

Keypad is an open source library that provides an event listener for a matrix keypad. It primarily uses functions to listen for changes of state of keypad buttons and returns this new state.

The LiquidCrystal library is inherent to the Arduino IDE and allows the Arduino to control LiquidCrystal dispays using a series of functions. It is written specifically for the Hitachi HD44780, or compatible, chipset with are used on most LCD units.

The program specification has been written that to comply with design, ten instances of each data type is used. The telegram function of the Modbus library, only allows for data type being unsigned16 bit integers. Therefor to allow for the digital data types, coils and digital inputs, unsigned 16 bit integers have been declared for each coil or digital, with only one bit of each integer being used. This is acknowledged as an inefficient use of memory.

When power is first applied to the microcontroller, the setup() function is run once. This contains the code to start the serial communication via the UART to the serial monitor application, creates the Modbus object using the SoftwareSerial and initiates the LCD screen with the introduction menus.

The main function loop() will continuously loop, listening for a keypad button to be activated. If it is, the keypad() function is called. Once the keypad() function returns, a large switch case control structure controls the flow of the program. The switch case structure can be summarised with the following points

- Write ten Holding Registers 40001 to 40010 using Function Code 16
- Write one Holding Register 400010 using Function Code 6
- Read ten Analogue Inputs 30001 to 30010 using Function Code 4
- Write to ten Coils 00001 to 00010 using Function Code 5 by using FC5 in ten instances
- Using the serial UART, writes the value of the above variables to the serial monitor on the connected (if connected) for troubleshooting and commissioning purposes

The keypad() function listens for activation of the following keys

- A
- B
- C
- D
- 0
- 1
- 2
- 3
- 5
- 6

If A is pressed the first Coil value, 00001 is displayed. Repeated pressing the A button will loop through all Coil values, displaying the value at that memory address until the loop is completed. The same applies for B, display Digital Input address and value, C, display Holding Register address and value, and D, display Analogue Input address and value.

If the 0 keypad button is pressed, all ten Coil addresses will be set False (0). If the 1 keypad button is pressed, all ten Coil addresses will be set True (1).

If the 2 keypad button is pressed all Holding Registers are preset to zero. If the 3 keypad button is pressed, the Holding Registers are preset to a range of values from 0 to 9000. If the 5 keypad button is pressed all ten Holding Registers are preset to 32767. Finally, if the 6 keypad button is pressed, the Holding Registers are all preset to 65535.

By pressing the * keypad button, the LCD returns to the main menu. On activation of any of the numeral keypad buttons, the display is updated to acknowledge what function has been executed and returns to the main menu. Keypad buttons, 4, 7, 8, 9 and # are free for use in future software versions.

Modbus communication parameters such as serial baud rate and Modbus Slave address ID are set in the Arduino program. The program code for the Arduino Modbus Simulator – Master can be found in Appendix C.1 The library file for ModbusRtu is found in Appendix C.3.

5.3. Arduino Modbus Simulator – Slave

After achieving the primary objective of the project specification by demonstrating a working Master solution, an extension to the scope was to develop a Slave. This required a new Modbus library as the previously selected was for a Master only. However, the reuse of the majority of the code from the Master solution was easily transferred. An open source Slave library chosen, Arduino-modbus-slave, with the source code found in Appendix C.4 to C.10. A difference between this Slave solution and the Master solution, is that a SoftwareSerial implementation is not possible. As the UART is needed for the Modbus serial communications, the serial monitor code for troubleshooting has been deleted.

The program for the Modbus Slave has been written in the Arduino IDE. The program structure can be broken down into the following sections

- Include Libraries
- Declaration of Variables
- Setup Code (ran once upon power initialisation)
- Main Code (ran in a continuous loop)
- Keypad Operation (a function called when the keypad is activated

There are six libraries used in this program being

- modbus
- modbusDevice
- modbusRegbank
- modbusSlave
- Keypad
- LiquidCrystal

The modbus, modbusDevice, modbusRegbank and modbusSlave libraries allows for communicating with Modbus Master devices using the serial interfaces, RS232, RS485 and USB, via the RTU message encoding protocol. It provides the query structure that allows the Slave to respond to the Master request query as per the Modbus specification.

Keypad is an open source library that provides an event listener for a matrix keypad. It primarily uses functions to listen for changes of state of keypad buttons and returns this new state.

The LiquidCrystal library is inherent to the Arduino IDE and allows the Arduino to control LiquidCrystal dispays using a series of functions. It is written specifically for the Hitachi HD44780, or compatible, chipset with are used on most LCD units.

The program specification has been written that to comply with design, ten instances of each data type is used. The library modbusRegbank is used to create the data type objects using the add function. 20 regBank instances are created with the following Modbus addressing

- Coils with address range 00001 to 00010
- Digital Inputs with address range 10001 to 10010
- Analogue Inputs with address range 30001 to 30010
- Holding Registers with address range 40001 to 40010

When power is first applied to the microcontroller, the setup() function is run once. This contains the code to create the regBank objects, setups the serial communication via the UART for Modbus, creates the Modbus object assigning the registers created and initiates the LCD screen with the introduction menus.

The main function loop() will continuously loop, listening for a keypad button to be activated. If it is, the keypad() function is called. Once the keypad() function returns, the Modbus slave function runs. The difference between the Master and the Slave solution, is the Slave only responds to the request command from the Master, therefore no additional code as per the large switch case in the Master software is needed.

The keypad() function listens for activation of the following keys

- A
- B
- C
- D
- 0
- 1
- 2
- 3
- 5
- 6

If A is pressed the first Coil value, 00001 is display. Repeat pressing A will loop through all Coil values, displaying the value at that memory address until the loop is completed. The same applies for B, display Digital Input address and value, C, display Holding Register address and value, and D, display Analogue Input address and value.

If the 0 keypad button is pressed, all ten Digital Input addresses will be set False (0). If the 1 keypad button is pressed, all ten Coil addresses will be set True (1).

If the 2 keypad button is pressed all Analogue Inputs are preset to zero. If the 3 keypad button is pressed, the Analogue Inputs are preset to a range of values from 0 to 9000. If the 5 keypad button is pressed all ten Analogue Inputs are preset to 32767. Finally, if the 6 keypad button is pressed, the Analogue Inputs are all preset to 65535.

By pressing the * keypad button, the LCD returns to the main menu. On activation of any of the numeral keypad buttons, the display is updated to acknowledge what function has been executed and returns to the main menu. Keypad buttons, 4, 7, 8, 9 and # are free for use in future software versions.

Modbus communication parameters such as serial baud rate and the Modbus Slave address ID are set in the Arduino program. The program code for the Arduino Modbus Simulator – Slave can be found in Appendix C.2.

6. Testing

The Arduino Modbus Simulator was tested against four difference scenarios. As a Master, it was tested with ModSim32 Slave, an Allen Bradley SLC500 Slave and an Invensys Triconex Slave. As a Slave it was tested with ModScan32 as the Master.

6.1. ModSim32 Slave

As represented by Figure 6.1, one of the test scenarios was to use the Arduino Modbus Simulator in Master configuration mode and use ModSim32 as the Slave device running on a Dell Precision laptop. Using RS232 as the physical transport layer, with a speed of 19200 baud, the Master used the Modbus request command structure to allow the ModSim32 Slave to respond.

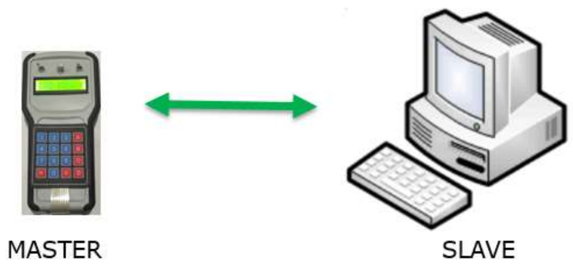


Figure 6.1: Arduino Modbus Simulator Master to ModSim32 Slave

In this test scenario, the Master used Function Code 16, write multiple registers, to write to ten Holding Registers in the Slave with the address range of 40001 to 40010. The data values were stepped from 0 to 32767 to 65535. The test was successful. The Arduino Modbus Simulator does not allow for a data value to be transmitted outside of the 0 to 65535 range. This is the limit of the unsigned integer data format.

For the second test, the Master used Function Code 6, write single register, to write to Holding Register address 40010. The data value was stepped 0 to 32767 to 65535. This test was successful.

The third test involved using Function Code 5, write single Coils, to change the values of Coils in the Slave with the address range of 00001 to 00010. The values were stepped from False (0) to True (1) successfully.

In the fourth test, the Master issued the request command Function Code 3 for the Slave to return the values of Holding Registers in the range of 40001 to 40010. The Slave successfully returned the value of these ten registers.

For the fifth and final test, the Master issued the request command Function Code 4 for the Slave to return the values of the ten Analogue Inputs in the address range of 30001 to 30010.

All test scenarios were repeated with the physical interface changed to RS485 and the outcome remained the same.

General testing included the stepping of the serial communications speed to the following values

- 9600 baud
- 19200 baud
- 38400 baud
- 56000 baud
- 115200 baud
- 256000 baud

In all speed configurations both the hardware and the software responded appropriately.

The testing of the Arduino Modbus Simulator as Modbus Master with ModSim32 as the Modbus Slave is considered a successful outcome.

6.2. Invensys Triconex Slave

To prove that the Arduino Modbus Simulator is an effective tool for use with industrial programmable controllers, two different controllers were selected for testing. The first of these controllers is an Invensys Triconex Safety Instrumented System, depicted in Figure 6.2.

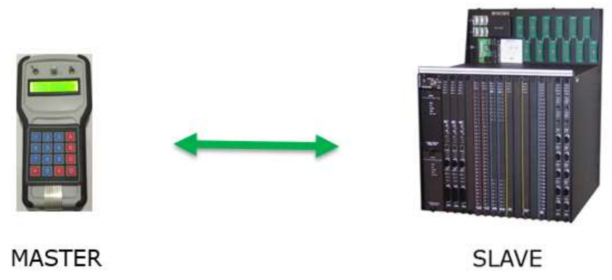


Figure 6.2: Arduino Modbus Simulator Master to Triconex Slave

A test program was developed for the Triconex with ten variables aliased to the Modbus addressing for Holding Registers 40001 to 40010. Ten Boolean variables were aliased to the Coils data type with address ranges 00001 to 00010. Both transport layers, RS-232 and RS-485, were used during testing. The test was successful and the Arduino Modbus Simulator can be used a Master with the Triconex.

6.3. SLC500 Slave

A test program for the SLC was developed for testing with the Arduino Modbus Simulator (Figure 6.2) with the following SLC data table mapped to Modbus addressing

- Coils: Word N10:9 bits 0 to 9 mapped to Modbus addresses 00001 to 00010
- Digital Inputs: Word N10:0 bits 0 to 9 mapped to Modbus addresses 10001 to 10010
- Holding Registers: Words N10:20 to N10:29 mapped to Modbus addresses 40001 to 40010
- Analogue Inputs: Words N10:10 to N10:19 mapped to Modbus addresses 30001 to 30010

As the SLC500 uses data tables, the use of Modbus addressing for Digital Inputs and Analogue Inputs was possible.

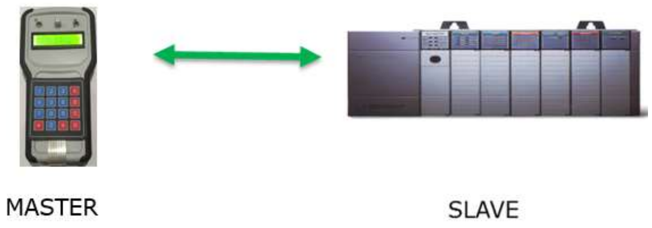


Figure 6.3: Arduino Modbus Simulator Master to SLC500 Slave

The testing for the SLC500 was completed successfully using both RS-232 and RS-485. Figure 6.3 shows the SLC500 with the RS-232 connection to the Arduino Modbus Simulator with RSLogix500 software in the background.

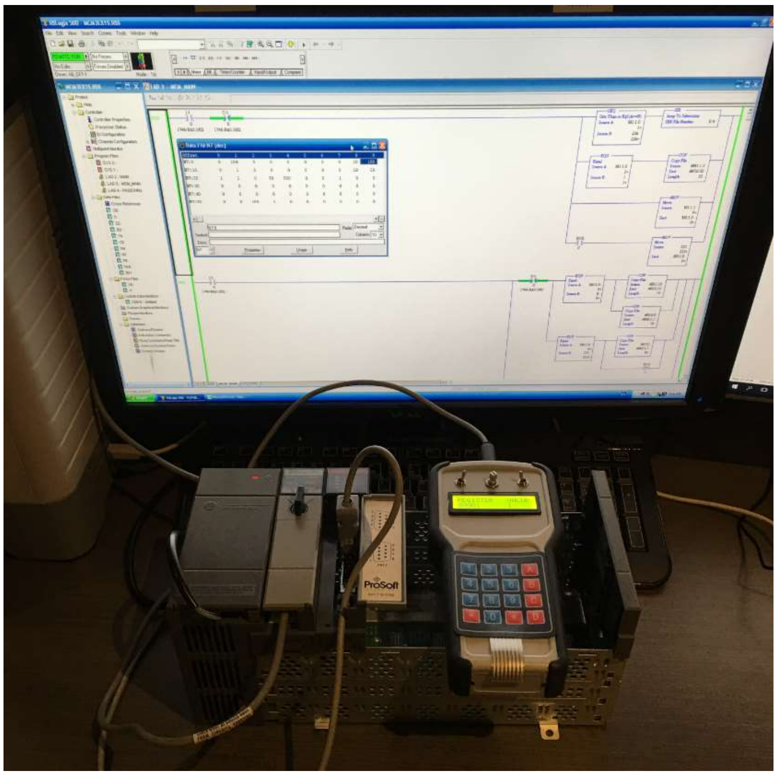


Figure 6.4: Arduino Modbus Simulator Master to SLC500 Slave

6.4. ModScan32 Master to Arduino Modbus Simulator Slave

As depicted in Figure 6.3, the final test programme was implemented using the ModScan32 as the Modbus and the Arduino Modbus Simulator as the Modbus Slave.

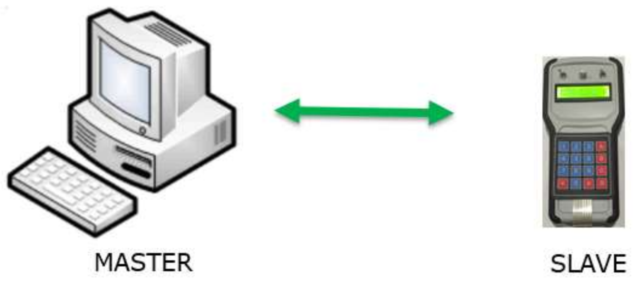


Figure 6.5: ModScan32 Master to Arduino Modbus Simulator Slave

For this test scenario, with ModScan32 running on a Dell Precision laptop, the Master used the Modbus request command structure to allow the Arduino Modbus Simulator Slave to respond. For the first test, in Modscan32, Holding Registers with the address range of 40001 to 40010 were manually adjusted with data values stepped from 0 to 32767 to 65535. The Arduino Modbus Simulator reflected these values by moving through the menu using keypad button C.

The second test involved setting the ModScan32 Coils values from False (0) to True (1) and back to False (0) for Modbus addresses 00001 to 00010. Using the keypad button A on the Arduino Modbus Simulator, each Coil address was highlighted and the value checked to confirm a match with the data value from the Master. This test was also a success.

To test that the Master could read the Digital Inputs and Analogue Inputs from the Arduino Modbus Simulator Slave, the following test routines were conducted.

For the Analogue Inputs in the Modbus address range of 30001 to 30010, using the keypad preprogramed options, the Arduino Modbus Simulator changed the values of each address from 0 to 32767 to 65535. The results were compared on the corresponding registers in ModScan32 and results matching. Another successful test.

The final test for the Arduino Modbus Simulator involved using the keypad to access the preprogramed functions to change the values of the Digital Inputs addressed 10001 to 10010 from False (0) then to True (1) and back to False (0). The corresponding addresses in ModScan32 reflected the same data values.

All test scenarios were repeated with the physical interface changed to RS485 and the outcome remained the same, a successful outcome.

6.5. Battery Test

A basic test was carried out on how long would the Arduino Modbus Simulator hold up under continuous use. The unit was connected via RS-232 to a Slave and executed the test program from chapter 6.1. Using a 9V Alkaline battery, a Duracell MN1604, the battery lasted 6 hours and ten minutes.

6.6. Incorrect Serial Connection

During the Project Preliminary Report for this disseration, negative consequential outcomes that could be caused by the Arduino Modbus Simulator were identified. One outcome was incorrectly plugging the wrong communication interface into a Slave Programmable Logic Controller would cause the PLC processor module to halt. A test was conducted where the Triconex was set up for RS-485 serial and the Arduino Modbus Simulator was setup for RS-232. While no communication was obviously possible, it did not cause the Tricon processor to go into fault mode. This was repeated on the SLC500 with same positive result. The scenario was also attempted with RS-232 on both PLCs and the Arduino Modbus Simulator set for RS-485, with no consequence. The Arduino Modbus Simulator hardware also suffered no effects from this, and once the matching transport layers were connected, communications were resumed.

7. Conclusions and Further Work

7.1. Conclusions

This project attempted to design and prototype an Arduino based Modbus Simulator with the goal of providing end users such as technicians and engineers a simple tool for use in commissioning and troubleshooting Modbus networks. The outcomes delivered by this project achieved the objective, as demonstrated by the content of this dissertation. From the evolution of a breadboard based prototype to the hand held unit weighing just 400 grams, the project has demonstrated a physical working unit. The Arduino Modbus Simulator is easy to use, provides the flexibility of both RS-232 and RS-485 serial communications and has been tested successfully with different industrial control systems.

7.2. Achievement of Project Objectives

The Project Specification objectives as demonstrated by Appendix A, have been addressed and either achieved or partially achieved. The numbered items below match the numbered items of the Project Specification with reference to sections of the dissertation to provide evidence of meeting the scope.

1. The research component has been completed by the literature review discussion in Chapter 2.

2. Chapter 3 which provides an analysis of several industrial Modbus networks meets the requirements by identifying and testing device capabilities.

3. Development of a hardware and software solution is covered by Chapters 4 and 5 respectively. Chapter 6 covers the testing of the Arduino Modbus Simulator to software application and two different control systems to prove diversity in testing.

4. The critical evaluation is partially covered by the test results in Chapter 6 and the Conclusions and Further Work discussion in Chapter 7.

5. Documented Code reference is provided in Appendix C. Appendix B displays the schematic and the User Manual is covered in Appendix D.

6. Three different enclosure options were considered until the final design selection. This is covered in Chapter 4.6. 3D printing was not investigated due to time restrictions.

7.3 Project Benefits

The project has highlighted three major benefits by producing the Arduino Modbus Simulator

- The concept for an open sourced Arduino based platform is possible and real. The project can be expanded beyond the proof of concept to a device that can be used as an alternative to expensive software solutions. The hardware used in the project device costs less than $100 dollars, while the cost of propriety software is $500 and laptop hardware to run this software costs at least $1000.

- The Arduino Modbus Simulator is light and ergonomic at 400 grams total weight. In comparison to the Dell Precision Laptop that was used during this project which weighs 7 kilograms. The Arduino Modbus Simulator wins this comparison.
- In any industry, when a Modbus communications link is down due to an unknown fault, the Mean Time To Repair costs production, safety or both. By having an easy to use tool to quickly divide and conquer will reduce this downtime and therefore have an economic or safety benefit.

7.4 Further Work

This project has demonstrated that is it possible to meet the project aim of an Arduino Modbus Simulator. However, there is further work that can be completed to take this prototype unit to a design that could compete with propriety software that runs on computer based hardware.

While the current design enables the user to select between RS-232 and RS-485 via a toggle switch, the actual serial communication setup is fixed at the time of the Arduino program download. To make this unit more flexible, the user menu to allow for dynamic setup of serial communication speed and Modbus Slave address is needed. The fixed addressing and number of datatypes also could be improved to allow the user to select dynamically via a menu. The final software improvement would allow the user to individually adjust coil or register values instead of using the pre-set commands.

The Arduino Modbus Simulator could also be expanded to include Modbus TCP as the transportation later in addition to serial communications.

A very challenging improvement would be to add other industrial protocols such as Profibus, HART and Fieldbus to the unit. If a single handheld unit was capable of being a multiple purpose communicator for commissioning and troubleshooting these industrial protocols, it would be a very popular product in industry.

References

Mackay, S, Wright, E, Park, J, Reynders, D 2004, *Practical Industrial Data Networks – Design, Installation and Troubleshooting,* Elsevier, Oxford

Modbus Organisation 2015, viewed October 2015, <http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>

Modbus Organisation 2015, viewed October 2015,

<http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf>

Modbus Organisation 2015, viewed October 2015,

<http://www.modbus.org/faq.php>

Niemirowski, G (2013), *Serial port and microcontrollers,* Charleston, United States of America

Axelson, J (1998), *Serial Port Complete,* Lakeview Research, United States of America

WinTech 2015, ModScan32, WinTech, Lewisburg, WV, USA

WinTech ModScan32, accessed April 2016

< http://www.win-tech.com/html/modbus1.htm>

WinTech ModSim32, accessed April 2016

< http://www.win-tech.com/html/modbus1.htm>

Modbus OPC Server for Modbus Devices, accessed June 2016

< https://www.matrikonopc.com/opc-drivers/opc-modbus/base-driver-details.aspx>

Real Time Automation 2015, *Modbus RTU Master Development Kit,* Wisconsin, United States

ModbusMaster, accessed 25 June 2016
<http://playground.arduino.cc/Code/ModbusMaster>

Modbus-arduino, accessed June 2016
<https://github.com/andresarmento/modbus-arduino>

simple-modbus, accessed June 2016
<https://code.google.com/archive/p/simple-modbus/>

Arduino-modbus-slave, accessed June 2016
<https://code.google.com/archive/p/arduino-modbus-slave/downloads>

Raspberry PI Modbus Master, accessed June 2016
<https://program-plc.blogspot.com.au/2014/10/modbus-rtu-communication-between-plc.html>

FreeModbus PIC, accessed June 2016
<http://www.opensourcepic.com/modbus.php>

QModBus, accessed June 2016
<https://sourceforge.net/projects/qmodbus/files/latest/download>

Jorgensen, P. C. (2014), *Software Testing – A Craftsman's Approach,* CRC Press, Taylor & Francis Groupm Florida, United States of America

Kaner, C, Falk, J, Nguyen, H (1999), *Testing Computer Software,* John Wiley & Sons, Canada

Patton, R (2006), *Software Testing,* Sams, Indiana, United States of America

Invensys Systems 2005, *Planning and Installation Guide,* United States

Aotewell 2016, viewed September 2016,
< http://www.aotewell.com/allen-bardley/allen-bradley-slc500-system.html>

Prosoft Technology, *3100-3150 MCM User Manual,* Bakersfield, United States

Blum, J, 2013, *Exploring Arduino,* Wiley, Indianapolis

Williams, E, 2014, *Make: AVR Programming,* MakerMedia, California

Hellenbuyck, C, *Beginner's Guide to Embedded C Programming Volume 3,* Electronic Products, Milford

Margolis, M, 2011, *Arduino Cookbook*, O'Reilly, California

ModbusRtu, accessed June 2016
< https://github.com/smarmengol/Modbus-Master-Slave-for-Arduino/blob/master/ModbusRtu.h>

Practical Troubleshooting and Problem Solving of Modbus and Modbus Plus Protocols, IDC Technologies

Appendix A Project Specification

Project Specification

For: Ryan Beccarelli

Title: Arduino Modbus Simulator

Major: Computer Systems Engineering

Supervisors: Catherine Hills Alexander Kist

Enrolment: Semester1&2 2016 EXTERNAL

Project Aim: Develop an Arduino based Modbus simulator to give end users such as technician s and engineers a new tool to use for commissioning and troubleshooting Modbus networks.

Programme:

1. Complete a literature review covering currently available Modbus hardware and software options, either open source or proprietary. This will include the physical standards for RS-232, RS-485 and TCP/IP networks, as well as the relevant protocol options for Modbus (RTU, ASCII and TCP). A review of the use of error checking mechanisms in these networks is also warranted.

2. Identify project requirements, in particular device capabilities with respect to network role, physical connections, protocol functions and error checking.

3. Develop a Hardware and Software solution to meet the objectives of the project requirements. Test the system for performance and record the results.

4. Undertake a validation of operation by reviewing performance results against the requirements analysis and complete a critical evaluation.

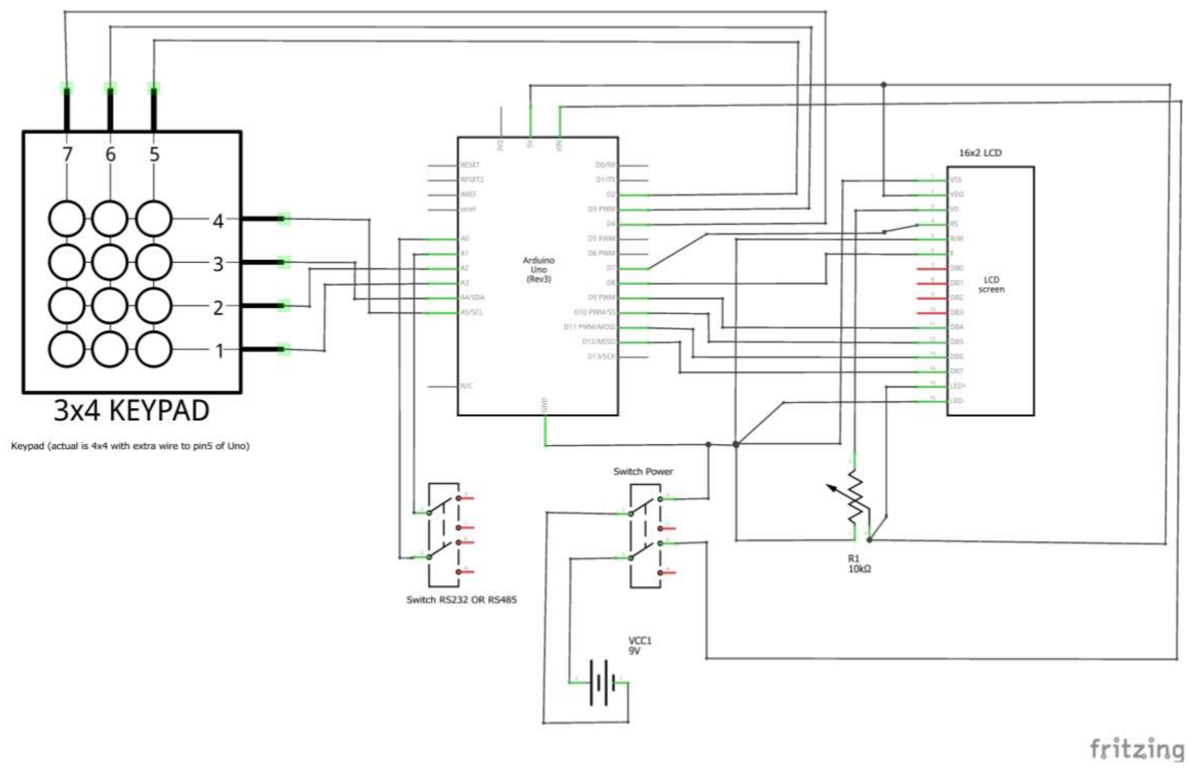
5. Create project documentation including a user manual, wiring / circuit schematics and documented code reference.

   If time permits

6. Develop an enclosure including to evaluate market available products for ergonomic and ruggedness prior to selection. Investigate a 3D printed enclosure for ergonomic and ruggedness and print a prototype.

Specification Approved Friday 8th April 2016

Appendix B Schematics



The wiring connection from the Arduino to the toggle switch (Switch RS232 or RS485) to the serial interfaces and then to the mini din connector is represented by the table below. Each entry represents the terminal number at the device.

Arduino	Switch	RS232	RS485	MINI DIN
	1		Tx	1
A0	2			
	3	Tx		3
	4		Rx	2
A1	5			
	6	Rx		4
5V		5V	5V	
GND		GND	GND	5

Appendix C Arduino Software

C.1 AMS_Master Code

```
/*
Arduino Modbus Simulator Version0
Written by Ryan Beccarelli
This program uses the ModbusRtu library provided under the GNU Free Software Foundation
This program also uses the Arduino Libraries
-SoftwareSerial
-Keypad
-LiquidCrystal
This program is designed to run on an Arduino Uno microcontroller
It uses the SoftwareSerial to interface to serial devices by RS-232 and RS-485
The program acts as a Modbus Master using the RTU message encoding format
It is configured to:
write ten holding registers using FC6 and FC16
read ten holding registers using FC3
read ten analogue input registers using FC4
write coils using FC5
Registers are setup for 10 Digital inputs but FC2 has not been implemented
The UART serial port is reserved for serial port monitoring by the connected computer
This allows for easy troubleshooting and commissioning
User input is provided by the 4x4 matrix keypad
Functions are preprogramed allowing for values to be set using keypad options
Local display is provided onto the 16x2 LCD display
*/
```

```
//Libraries
#include <ModbusRtu.h>
#include <SoftwareSerial.h>
#include <Keypad.h>
#include <LiquidCrystal.h>

LiquidCrystal lcd(7,8,9,10,11,12); // Allocate pins to LCD



// Registers for Modbus Master
uint8_t u8state;

// 0x Status Coils
uint16_t W_Coils0[16];
uint16_t W_Coils1[16];
uint16_t W_Coils2[16];
uint16_t W_Coils3[16];
uint16_t W_Coils4[16];
uint16_t W_Coils5[16];
uint16_t W_Coils6[16];
uint16_t W_Coils7[16];
uint16_t W_Coils8[16];
uint16_t W_Coils9[16];

// 1x Input Coils
uint16_t R_Coils0[16];
uint16_t R_Coils1[16];
uint16_t R_Coils2[16];
```

```
uint16_t R_Coils3[16];
uint16_t R_Coils4[16];
uint16_t R_Coils5[16];
uint16_t R_Coils6[16];
uint16_t R_Coils7[16];
uint16_t R_Coils8[16];
uint16_t R_Coils9[16];



// 3x Input Registers
uint16_t R_Registers[16];

// 4x Holding Registers
uint16_t W_Registers[16];

// Registers for LCD operations
char operation,button;
int Acount=1;
int Bcount=1;
int Ccount=1;
int Dcount=1;

// Registers for keypad operations
const byte ROWS = 4;
const byte COLS = 4;

char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};

byte rowPins[ROWS] = {A2,A3,A4,A5}; //connect to the row pinouts of the keypad
byte colPins[COLS] = {2,3,4,5}; //connect to the column pinouts of the keypad
Keypad customKeypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS);



/**
 *  Modbus object declaration
 *  u8id : node id = 0 for master, = 1..247 for slave
 *  u8serno : serial port (use 0 for Serial)
 *  u8txenpin : 0 for RS-232 and USB-FTDI
 *               or any pin number > 1 for RS-485
 */
Modbus master(0); // this is master and RS-232 or USB-FTDI via software serial


//This is an structe which contains a query to an slave device
modbus_t telegram;

unsigned long u32wait;

//Creates a SoftwareSerial object so that we can use software serial
SoftwareSerial mySerial(A1, A0);
```

```
void setup() {

  //Start of Setup Code for Modbus Master
  Serial.begin(9600);//use the hardware serial if you want to connect to your computer via usb
cable, etc.
  master.begin( &mySerial, 19200 ); // begin the ModBus object. The first parameter is the
address of the SoftwareSerial address
  master.setTimeOut( 2000 ); // if there is no answer in 2000 ms, roll over
  u32wait = millis() + 1000;
  u8state = 0;
  // End of Setup Code for Modbus Master

  // Setup code for serial monitoring to computer
  Serial.begin(9600);
  // Setup Code for LCD & Keypad
  lcd.begin(16,2); // initialize the lcd
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Arduino Modbus");
  lcd.setCursor(0,1);
  lcd.print("Simulator Master");
  delay(4000);
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("By R.Beccarelli");
  lcd.setCursor(0,1);
  lcd.print("Menu Loading");
  delay(4000);
```

```
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");
 // End of Setup Code for LCD & Keypad
}

void loop() {

 //Code for Keypad & LCD
 button = customKeypad.getKey(); // Button read
    if (button=='*') // Return to main menu
    {
     lcd.clear();
     lcd.setCursor(0,0);
     lcd.print("A=FC01 B=FC02");
     lcd.setCursor(0,1);
     lcd.print("C=FC03 D=FC04");
    }

    if ((button=='A' || button=='B' || button=='C' || button=='D' || button=='0' || button=='1'
    || button=='2' || button=='3' || button=='5' || button=='6')) // If user is done inputing numbers
    {
     operation = button; // operation remembers what register operation user wants on to view
     keypad();
    }
 // End of code for Keypad & LCD
 // Code for Modbus Master
 switch( u8state ) {
 case 0:
   if (millis() > u32wait) u8state++; // wait state
   break;

 //Read 40001 to 40010 using FC3
 case 1:
   telegram.u8id = 1; // slave address
   telegram.u8fct = 3; // function code (this one is registers read)
   telegram.u16RegAdd = 0; // start address in slave
   telegram.u16CoilsNo = 10; // number of elements (coils or registers) to read
   telegram.au16reg = R_Registers;
   master.query( telegram ); // send query (only once)
   u8state++;
   break;
 case 2:
   master.poll(); // check incoming messages
   if (master.getState() == COM_IDLE) {
     u8state++;
     u32wait = millis() + 50;
    //Send to serial monitor for troubleshooting/commissioning
    Serial.println("40001 to 40010");
    Serial.println(W_Registers[0]);
    Serial.println(W_Registers[1]);
    Serial.println(W_Registers[2]);
     Serial.println(W_Registers[3]);
     Serial.println(W_Registers[4]);
     Serial.println(W_Registers[5]);
     Serial.println(W_Registers[6]);
     Serial.println(W_Registers[7]);
     Serial.println(W_Registers[8]);
     Serial.println(W_Registers[9]);
   }
   break;

 //Write 40001 to 40010 using FC16
 case 3:
   telegram.u8id = 1; // slave address
   telegram.u8fct = 16; // function code (this one is registers read)
   telegram.u16RegAdd = 0; // start address in slave
   telegram.u16CoilsNo = 10; // number of elements (coils or registers) to read
   //telegram.au16reg = au16data; // pointer to a memory array in the Arduino
   telegram.au16reg = W_Registers;
   master.query( telegram ); // send query (only once)
   u8state++;
   break;
 case 4:
   master.poll(); // check incoming messages
   if (master.getState() == COM_IDLE) {
     u8state++;
     u32wait = millis() + 50;
   }
   break;
```

```
//Write 40010 using FC6
case 5:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 6; // function code (this one is registers read)
  telegram.u16RegAdd = 9; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Registers;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;
case 6:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Read Analog Inputs 30001 to 30010 using FC4
case 7:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 4; // function code (this one is registers read)
  telegram.u16RegAdd = 0; // start address in slave
  telegram.u16CoilsNo = 10; // number of elements (coils or registers) to read
  //telegram.au16reg = au16data; // pointer to a memory array in the Arduino
  telegram.au16reg = R_Registers;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;
case 8:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
      //Send to serial monitor for troubleshooting/commissioning
      Serial.println("30001 to 30010");
      Serial.println(R_Registers[0]);
      Serial.println(R_Registers[1]);
      Serial.println(R_Registers[2]);
      Serial.println(R_Registers[3]);
      Serial.println(R_Registers[4]);
      Serial.println(R_Registers[5]);
      Serial.println(R_Registers[6]);
      Serial.println(R_Registers[7]);
      Serial.println(R_Registers[8]);
      Serial.println(R_Registers[9]);
  }
  break;

//Write Coil 00001
case 9:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 0; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  //telegram.au16reg = au16data; // pointer to a memory array in the Arduino
  telegram.au16reg = W_Coils0;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

case 10:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00002
case 11:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 1; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils1;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

  case 12:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    //u8state=0;
    u8state++;
    u32wait = millis() + 50;

  }
  break;

//Write Coil 00003
case 13:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 2; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils2;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

  case 14:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00004
case 15:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 3; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils3;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;
  case 16:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00005
case 17:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 4; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils4;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;


  case 18:
```

```
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00006
case 19:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 5; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils5;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

  case 20:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00007
case 21:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 6; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils6;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

  case 22:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00008
case 23:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 7; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils7;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;
  case 24:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00009
case 25:
  telegram.u8id = 1; // slave address
  telegram.u8fct = 5; // function code (this one is registers read)
  telegram.u16RegAdd = 8; // start address in slave
  telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
  telegram.au16reg = W_Coils8;
  master.query( telegram ); // send query (only once)
  u8state++;
  break;

  case 26:
  master.poll(); // check incoming messages
  if (master.getState() == COM_IDLE) {
    u8state++;
    u32wait = millis() + 50;
  }
  break;

//Write Coil 00010
```

```
case 27:
 telegram.u8id = 1; // slave address
 telegram.u8fct = 5; // function code (this one is registers read)
 telegram.u16RegAdd = 9; // start address in slave
 telegram.u16CoilsNo = 1; // number of elements (coils or registers) to read
 telegram.au16reg = W_Coils9;
 master.query( telegram ); // send query (only once)
 u8state++;
 break;

 case 28:
 master.poll(); // check incoming messages
 if (master.getState() == COM_IDLE) {
  u8state=0;
  u32wait = millis() + 50;
  //Send to serial monitor for troubleshooting/commissioning
  Serial.println("00001 to 00010");
  Serial.println(W_Coils0[0]);
  Serial.println(W_Coils1[0]);
  Serial.println(W_Coils2[0]);
  Serial.println(W_Coils3[0]);
  Serial.println(W_Coils4[0]);
  Serial.println(W_Coils5[0]);
  Serial.println(W_Coils6[0]);
  Serial.println(W_Coils7[0]);
  Serial.println(W_Coils8[0]);
  Serial.println(W_Coils9[0]);
  Serial.println("10001 to 10010");
```

```
      Serial.println(R_Coils0[0]);
      Serial.println(R_Coils1[0]);
      Serial.println(R_Coils2[0]);
      Serial.println(R_Coils3[0]);
      Serial.println(R_Coils4[0]);
      Serial.println(R_Coils5[0]);
      Serial.println(R_Coils6[0]);
      Serial.println(R_Coils7[0]);
      Serial.println(R_Coils8[0]);
      Serial.println(R_Coils9[0]);
    }
    break;
  }
  // End of Code for Modbus Master
}

// Function for Keypad Operation
void keypad() // Simple switch case to pick what register to view, based on button pressed by
user.
{
  switch(operation)
    {
      case 'A': // Addition
          lcd.clear();
          lcd.setCursor(0,0);
          lcd.print("REGISTER   VALUE");

          if (Acount == 1)
          {
```

```
   lcd.setCursor(0,1);
   lcd.print("00001");
   lcd.setCursor(11,1);
   W_Coils0[0]=0;
   lcd.print(W_Coils0[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 2)
{
   lcd.setCursor(0,1);
   lcd.print("00001");
   lcd.setCursor(11,1);
   W_Coils0[0]=1;
   lcd.print(W_Coils0[0]);
   Serial.println (Acount);
   //Acount++;
   Acount=1;
}
else if (Acount == 3)
{
   lcd.setCursor(0,1);
   lcd.print("00003");
   lcd.setCursor(11,1);
   lcd.print(W_Coils2[0]);
   Serial.println (Acount);
   Acount++;
}
```

```
else if (Acount == 4)
{
   lcd.setCursor(0,1);
   lcd.print("00004");
   lcd.setCursor(11,1);
   lcd.print(W_Coils3[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 5)
{
   lcd.setCursor(0,1);
   lcd.print("00005");
   lcd.setCursor(11,1);
   lcd.print(W_Coils4[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("00006");
   lcd.setCursor(11,1);
   lcd.print(W_Coils5[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 7)
```

```
{
   lcd.setCursor(0,1);
   lcd.print("00007");
   lcd.setCursor(11,1);
   lcd.print(W_Coils6[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 8)
{
   lcd.setCursor(0,1);
   lcd.print("00008");
   lcd.setCursor(11,1);
   lcd.print(W_Coils7[0]);
   Serial.println (Acount);
   Acount++;
}
else if (Acount == 9)
{
   lcd.setCursor(0,1);
   lcd.print("00009");
   lcd.setCursor(11,1);
   lcd.print(W_Coils8[0]);
   Serial.println (Acount);
   Acount++;
}
else
{
        lcd.setCursor(0,1);
        lcd.print("00010");
        lcd.setCursor(11,1);
        lcd.print(W_Coils9[0]);
        Serial.println (Acount);
        Acount=1;
    }
    break;

case 'B': //
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("REGISTER   VALUE");

    if (Bcount == 1)
    {
        lcd.setCursor(0,1);
        lcd.print("10001");
        lcd.setCursor(11,1);
        lcd.print(R_Coils0[0]);
        Serial.println (Bcount);
        Bcount++;
    }
    else if (Bcount == 2)
    {
        lcd.setCursor(0,1);
        lcd.print("10002");
        lcd.setCursor(11,1);
    lcd.print(R_Coils1[0]);
    Serial.println (Bcount);
    Bcount++;
}
else if (Bcount == 3)
{
    lcd.setCursor(0,1);
    lcd.print("10003");
    lcd.setCursor(11,1);
    lcd.print(R_Coils2[0]);
    Serial.println (Bcount);
    Bcount++;
}
else if (Bcount == 4)
{
    lcd.setCursor(0,1);
    lcd.print("10004");
    lcd.setCursor(11,1);
    lcd.print(R_Coils3[0]);
    Serial.println (Bcount);
    Bcount++;
}
else if (Bcount == 5)
{
    lcd.setCursor(0,1);
    lcd.print("10005");
    lcd.setCursor(11,1);
    lcd.print(R_Coils4[0]);
```

```
   Serial.println (Bcount);
   Bcount++;
}
else if (Bcount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("10006");
   lcd.setCursor(11,1);
   lcd.print(R_Coils5[0]);
   Serial.println (Bcount);
   Bcount++;
}
else if (Bcount == 7)
{
   lcd.setCursor(0,1);
   lcd.print("10007");
   lcd.setCursor(11,1);
   lcd.print(R_Coils6[0]);
   Serial.println (Bcount);
   Bcount++;
}
else if (Bcount == 8)
{
   lcd.setCursor(0,1);
   lcd.print("10008");
   lcd.setCursor(11,1);
   lcd.print(R_Coils7[0]);
   Serial.println (Bcount);
```

```
        Bcount++;
      }
      else if (Bcount == 9)
      {
        lcd.setCursor(0,1);
        lcd.print("10009");
        lcd.setCursor(11,1);
        lcd.print(R_Coils8[0]);
        Serial.println (Bcount);
        Bcount++;
      }
      else
      {
        lcd.setCursor(0,1);
        lcd.print("10010");
        lcd.setCursor(11,1);
        lcd.print(R_Coils9[0]);
        Serial.println (Bcount);
        Bcount=1;
      }
      break;

    case 'C':
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("REGISTER   VALUE");

      if (Ccount == 1)
```

```
{
   lcd.setCursor(0,1);
   lcd.print("40001");
   lcd.setCursor(11,1);
   lcd.print(W_Registers[0]);
   Serial.println (Ccount);
   Ccount++;
}
else if (Ccount == 2)
{
   lcd.setCursor(0,1);
   lcd.print("40002");
   lcd.setCursor(11,1);
   lcd.print(W_Registers[1]);
   Serial.println (Ccount);
   Ccount++;
}
else if (Ccount == 3)
{
   lcd.setCursor(0,1);
   lcd.print("40003");
   lcd.setCursor(11,1);
   lcd.print(W_Registers[2]);
   Serial.println (Ccount);
   Ccount++;
}
else if (Ccount == 4)
{
    lcd.setCursor(0,1);
    lcd.print("40004");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[3]);
    Serial.println (Ccount);
    Ccount++;
}
else if (Ccount == 5)
{
    lcd.setCursor(0,1);
    lcd.print("40005");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[4]);
    Serial.println (Ccount);
    Ccount++;
}
else if (Ccount == 6)
{
    lcd.setCursor(0,1);
    lcd.print("40006");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[5]);
    Serial.println (Ccount);
    Ccount++;
}
else if (Ccount == 7)
{
    lcd.setCursor(0,1);
    lcd.print("40007");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[6]);
    Serial.println (Ccount);
    Ccount++;
}
else if (Ccount == 8)
{
    lcd.setCursor(0,1);
    lcd.print("40008");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[7]);
    Serial.println (Ccount);
    Ccount++;
}
else if (Ccount == 9)
{
    lcd.setCursor(0,1);
    lcd.print("40009");
    lcd.setCursor(11,1);
    lcd.print(W_Registers[8]);
    Serial.println (Ccount);
    Ccount++;
}
else
{
    lcd.setCursor(0,1);
    lcd.print("40010");
```

```
        lcd.setCursor(11,1);
        lcd.print(W_Registers[9]);
        Serial.println (Ccount);
        Ccount=1;
    }
    break;

case 'D':
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("REGISTER   VALUE");

    if (Dcount == 1)
    {
        lcd.setCursor(0,1);
        lcd.print("30001");
        lcd.setCursor(11,1);
        lcd.print(R_Registers[0]);
        Serial.println (Dcount);
        Dcount++;
    }
    else if (Dcount == 2)
    {
        lcd.setCursor(0,1);
        lcd.print("30002");
        lcd.setCursor(11,1);
        lcd.print(R_Registers[1]);
        Serial.println (Dcount);
    Dcount++;
}
else if (Dcount == 3)
{
    lcd.setCursor(0,1);
    lcd.print("30003");
    lcd.setCursor(11,1);
    lcd.print(R_Registers[2]);
    Serial.println (Dcount);
    Dcount++;
}
else if (Dcount == 4)
{
    lcd.setCursor(0,1);
    lcd.print("30004");
    lcd.setCursor(11,1);
    lcd.print(R_Registers[3]);
    Serial.println (Dcount);
    Dcount++;
}
else if (Dcount == 5)
{
    lcd.setCursor(0,1);
    lcd.print("30005");
    lcd.setCursor(11,1);
    lcd.print(R_Registers[4]);
    Serial.println (Dcount);
    Dcount++;
```

```
}
else if (Dcount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("30006");
   lcd.setCursor(11,1);
   lcd.print(R_Registers[5]);
   Serial.println (Dcount);
   Dcount++;
}
else if (Dcount == 7)
{
   lcd.setCursor(0,1);
   lcd.print("30007");
   lcd.setCursor(11,1);
   lcd.print(R_Registers[6]);
   Serial.println (Dcount);
   Dcount++;
}
else if (Dcount == 8)
{
   lcd.setCursor(0,1);
   lcd.print("30008");
   lcd.setCursor(11,1);
   lcd.print(R_Registers[7]);
   Serial.println (Dcount);
   Dcount++;
}
```

```
 else if (Dcount == 9)
 {
    lcd.setCursor(0,1);
    lcd.print("30009");
    lcd.setCursor(11,1);
    lcd.print(R_Registers[8]);
    Serial.println (Dcount);
    Dcount++;
 }
 else
 {
    lcd.setCursor(0,1);
    lcd.print("30010");
    lcd.setCursor(11,1);
    lcd.print(R_Registers[9]);
    Serial.println (Dcount);
    Dcount=1;
 }
 break;

case '0':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("All Coils");
 lcd.setCursor(0,1);
 lcd.print("Set To False");

 //Sets all coils to FALSE
 W_Coils0[0]=0;
 W_Coils1[0]=0;
 W_Coils2[0]=0;
 W_Coils3[0]=0;
 W_Coils4[0]=0;
 W_Coils5[0]=0;
 W_Coils6[0]=0;
 W_Coils7[0]=0;
 W_Coils8[0]=0;
 W_Coils9[0]=0;

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");

break;

case '1':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("All Coils");
 lcd.setCursor(0,1);
 lcd.print("Set To True");

 //Sets all coils to TRUE
 W_Coils0[0]=1;
 W_Coils1[0]=1;
 W_Coils2[0]=1;
 W_Coils3[0]=1;
 W_Coils4[0]=1;
 W_Coils5[0]=1;
 W_Coils6[0]=1;
 W_Coils7[0]=1;
 W_Coils8[0]=1;
 W_Coils9[0]=1;

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");

break;

case '2':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
 lcd.print("Set To Zero");

 //Sets all Holding Registers to Zero
 W_Registers[0]=0;
 W_Registers[1]=0;
 W_Registers[2]=0;
 W_Registers[3]=0;
 W_Registers[4]=0;
 W_Registers[5]=0;
 W_Registers[6]=0;
 W_Registers[7]=0;
 W_Registers[8]=0;
 W_Registers[9]=0;

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");



break;

case '3':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
 lcd.print("HR# * 10^3");
 //Test numbers for Holding Registers
 W_Registers[0]=0;
 W_Registers[1]=1000;
 W_Registers[2]=2000;
 W_Registers[3]=3000;
 W_Registers[4]=4000;
 W_Registers[5]=5000;
 W_Registers[6]=6000;
 W_Registers[7]=7000;
 W_Registers[8]=8000;
 W_Registers[9]=9000;

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");



break;

case '5':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
 lcd.print("Set To 32767");
 //Test numbers for Holding Registers
 W_Registers[0]=32767;
 W_Registers[1]=32767;
 W_Registers[2]=32767;
 W_Registers[3]=32767;
 W_Registers[4]=32767;
 W_Registers[5]=32767;
 W_Registers[6]=32767;
 W_Registers[7]=32767;
 W_Registers[8]=32767;
 W_Registers[9]=32767;

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");


break;

case '6':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
```

```
          lcd.print("Set To 65535");

          //Test numbers for Holding Registers
          W_Registers[0]=65535;
          W_Registers[1]=65535;
          W_Registers[2]=65535;
          W_Registers[3]=65535;
          W_Registers[4]=65535;
          W_Registers[5]=65535;
          W_Registers[6]=65535;
          W_Registers[7]=65535;
          W_Registers[8]=65535;
          W_Registers[9]=65535;

          delay(2000);
          lcd.clear();
          lcd.setCursor(0,0);
          lcd.print("A=FC01 B=FC02");
          lcd.setCursor(0,1);
          lcd.print("C=FC03 D=FC04");
         break;
    }
}
```

C.2 AMS_Slave Code

```
/*
Ardunio Modbus Simulator Version0
```

```
Written by Ryan Beccarelli
This program uses the Modbus library provided under the GNU Free Software Foundation
This program also uses the Arduino Libraries
-SoftwareSerial
-Keypad
-LiquidCrystal
This program is designed to run on an Arduino Uno microcontroller
The UART serial port is used for serial communication and needs to be disconnected when up
loading
from the Arduino IDE
The program acts as a Modbus slave using the RTU message encoding format
It is configured with the folliwng memory:
10 Coils for writing to from the Master
10 Digital Inputs that can be read by the Master
10 Holding Registers for writing to from the Master
10 Analogue Inputs that can be read by the Master
User input is provided by the 4x4 matrix keypad
Functions are preprogrammed allowing for values to be set using keypad options
Local dispay is provided onto the 16x2 LCD display
*/


//Libraries
#include <modbus.h>
#include <modbusDevice.h>
#include <modbusRegBank.h>
#include <modbusSlave.h>
#include <SoftwareSerial.h>
#include <Keypad.h>
#include <LiquidCrystal.h>
```

```
LiquidCrystal lcd(7,8,9,10,11,12); // Allocate pins to LCD


// Registers for Modbus Master
modbusDevice regBank;
modbusSlave slave;



// Registers for LCD operations
char operation,button;
int Acount=1;
int Bcount=1;
int Ccount=1;
int Dcount=1;

// Registers for keypad operations
const byte ROWS = 4;
const byte COLS = 4;

char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
```

```
byte rowPins[ROWS] = {A2,A3,A4,A5}; //connect to the row pinouts of the keypad
byte colPins[COLS] = {2,3,4,5}; //connect to the column pinouts of the keypad
Keypad customKeypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS);



void setup() {

  regBank.setId(1); ///Set Slave ID
//Add Digital Input registers
// 1x Input Coils
  regBank.add(10001);
  regBank.add(10002);
  regBank.add(10003);
  regBank.add(10004);
  regBank.add(10005);
  regBank.add(10006);
  regBank.add(10007);
  regBank.add(10008);
  regBank.add(10009);
  regBank.add(10010);

// Add Coils
// 0x Status Coils
  regBank.add(1);
  regBank.add(2);
  regBank.add(3);
  regBank.add(4);
  regBank.add(5);
  regBank.add(6);
  regBank.add(7);
  regBank.add(8);
  regBank.add(9);
  regBank.add(10);

//Analog input registers
// 3x Input Registers
  regBank.add(30001);
  regBank.add(30002);
  regBank.add(30003);
  regBank.add(30004);
  regBank.add(30005);
  regBank.add(30006);
  regBank.add(30007);
  regBank.add(30008);
  regBank.add(30009);
  regBank.add(30010);

//Holding Registers
// 4x Holding Registers
  regBank.add(40001);
  regBank.add(40002);
  regBank.add(40003);
  regBank.add(40004);
  regBank.add(40005);
  regBank.add(40006);
  regBank.add(40007);
  regBank.add(40008);
  regBank.add(40009);
  regBank.add(40010);


  //Setup the Modbus communication link
  slave._device = &regBank;
  slave.setBaud(19200);


  // Setup Code for LCD & Keypad
  lcd.begin(16,2); // initialize the lcd
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Arduino Modbus");
  lcd.setCursor(0,1);
  lcd.print("Simulator Slave");
  delay(4000);
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("By R.Beccarelli");
  lcd.setCursor(0,1);
  lcd.print("Menu Loading");
```

```
  delay(4000);
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("A=FC01 B=FC02");
  lcd.setCursor(0,1);
  lcd.print("C=FC03 D=FC04");
  // End of Setup Code for LCD & Keypad
}

void loop() {

  //Code for Keypad & LCD
  button = customKeypad.getKey(); // Button read
      if (button=='*') // Return to main menu
      {
        lcd.clear();
        lcd.setCursor(0,0);
        lcd.print("A=FC01 B=FC02");
        lcd.setCursor(0,1);
        lcd.print("C=FC03 D=FC04");
      }

      if ((button=='A' || button=='B' || button=='C' || button=='D' || button=='0' || button=='1'
      || button=='2' || button=='3' || button=='5' || button=='6')) // If user is done inputing numbers
      {
        operation = button; // operation remembers what register operation user wants on to view
        keypad();
      }
```

```
 // End of code for Keypad & LCD



 slave.run();



}

// Function for Keypad Operation
void keypad() // Simple switch case to pick what register to view, based on button pressed by
user.
{
 switch(operation)
   {
     case 'A': // Addition
        lcd.clear();
        lcd.setCursor(0,0);
        lcd.print("REGISTER   VALUE");

        if (Acount == 1)
        {
           lcd.setCursor(0,1);
           lcd.print("00001");
           lcd.setCursor(11,1);
           lcd.print(regBank.get(1));
           Acount++;
        }
        else if (Acount == 2)
        {
```

```
   lcd.setCursor(0,1);
   lcd.print("00002");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(2));
   Acount++;

}
else if (Acount == 3)
{
   lcd.setCursor(0,1);
   lcd.print("00003");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(3));
   Acount++;
}
else if (Acount == 4)
{
   lcd.setCursor(0,1);
   lcd.print("00004");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(4));
   Acount++;
}
else if (Acount == 5)
{
   lcd.setCursor(0,1);
   lcd.print("00005");
   lcd.setCursor(11,1);
```

```
    lcd.print(regBank.get(5));
    Acount++;
}
else if (Acount == 6)
{
    lcd.setCursor(0,1);
    lcd.print("00006");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(6));
    Acount++;
}
else if (Acount == 7)
{
    lcd.setCursor(0,1);
    lcd.print("00007");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(7));
    Acount++;
}
else if (Acount == 8)
{
    lcd.setCursor(0,1);
    lcd.print("00008");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(8));
    Acount++;
}
else if (Acount == 9)
```

```
    {
      lcd.setCursor(0,1);
      lcd.print("00009");
      lcd.setCursor(11,1);
      lcd.print(regBank.get(9));
      Acount++;
    }
    else
    {
      lcd.setCursor(0,1);
      lcd.print("00010");
      lcd.setCursor(11,1);
      lcd.print(regBank.get(10));
      Acount=1;
    }
    break;

  case 'B': //
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("REGISTER   VALUE");

    if (Bcount == 1)
    {
      lcd.setCursor(0,1);
      lcd.print("10001");
      lcd.setCursor(11,1);
      lcd.print(regBank.get(10001));
```

```
   Bcount++;
}
else if (Bcount == 2)
{
   lcd.setCursor(0,1);
   lcd.print("10002");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10002));
   Bcount++;
}
else if (Bcount == 3)
{
   lcd.setCursor(0,1);
   lcd.print("10003");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10003));
   Bcount++;
}
else if (Bcount == 4)
{
   lcd.setCursor(0,1);
   lcd.print("10004");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10004));
   Bcount++;
}
else if (Bcount == 5)
{
   lcd.setCursor(0,1);
   lcd.print("10005");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10005));
   Bcount++;
}
else if (Bcount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("10006");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10006));
   Bcount++;
}
else if (Bcount == 7)
{
   lcd.setCursor(0,1);
   lcd.print("10007");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10007));
   Bcount++;
}
else if (Bcount == 8)
{
   lcd.setCursor(0,1);
   lcd.print("10008");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(10008));
```

```
        Bcount++;
    }
    else if (Bcount == 9)
    {
        lcd.setCursor(0,1);
        lcd.print("10009");
        lcd.setCursor(11,1);
        lcd.print(regBank.get(10009));
        Bcount++;
    }
    else
    {
        lcd.setCursor(0,1);
        lcd.print("10010");
        lcd.setCursor(11,1);
        lcd.print(regBank.get(10010));
        Bcount=1;
    }
    break;

case 'C':
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("REGISTER   VALUE");

    if (Ccount == 1)
    {
        lcd.setCursor(0,1);
    lcd.print("40001");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(40001));
    Ccount++;
}
else if (Ccount == 2)
{
    lcd.setCursor(0,1);
    lcd.print("40002");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(40002));
    Ccount++;
}
else if (Ccount == 3)
{
    lcd.setCursor(0,1);
    lcd.print("40003");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(40003));
    Ccount++;
}
else if (Ccount == 4)
{
    lcd.setCursor(0,1);
    lcd.print("40004");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(40004));
    Ccount++;
```

```
}
else if (Ccount == 5)
{
   lcd.setCursor(0,1);
   lcd.print("40005");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(40005));
   Ccount++;
}
else if (Ccount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("40006");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(40006));
   Ccount++;
}
else if (Ccount == 7)
{
   lcd.setCursor(0,1);
   lcd.print("40007");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(40007));
   Ccount++;
}
else if (Ccount == 8)
{
   lcd.setCursor(0,1);
```

```
        lcd.print("40008");
        lcd.setCursor(11,1);
        lcd.print(regBank.get(40008));
        Ccount++;
    }
    else if (Ccount == 9)
    {
        lcd.setCursor(0,1);
        lcd.print("40009");
        lcd.setCursor(11,1);
        lcd.print(regBank.get(40009));
        Ccount++;
    }
    else
    {
        lcd.setCursor(0,1);
        lcd.print("40010");
        lcd.setCursor(11,1);
        lcd.print(regBank.get(40010));
        Ccount=1;
    }
    break;

case 'D':
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("REGISTER   VALUE");
```

```
if (Dcount == 1)
{
   lcd.setCursor(0,1);
   lcd.print("30001");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30001));
   Dcount++;
}
else if (Dcount == 2)
{
   lcd.setCursor(0,1);
   lcd.print("30002");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30002));
   Dcount++;
}
else if (Dcount == 3)
{
   lcd.setCursor(0,1);
   lcd.print("30003");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30003));
   Dcount++;
}
else if (Dcount == 4)
{
   lcd.setCursor(0,1);
   lcd.print("30004");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30004));
   Dcount++;
}
else if (Dcount == 5)
{
   lcd.setCursor(0,1);
   lcd.print("30005");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30005));
   Dcount++;
}
else if (Dcount == 6)
{
   lcd.setCursor(0,1);
   lcd.print("30006");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30006));
   Dcount++;
}
else if (Dcount == 7)
{
   lcd.setCursor(0,1);
   lcd.print("30007");
   lcd.setCursor(11,1);
   lcd.print(regBank.get(30007));
   Dcount++;
}
```

```
 else if (Dcount == 8)
 {
    lcd.setCursor(0,1);
    lcd.print("30008");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(30008));
    Dcount++;
 }
 else if (Dcount == 9)
 {
    lcd.setCursor(0,1);
    lcd.print("30009");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(30009));
    Dcount++;
 }
 else
 {
    lcd.setCursor(0,1);
    lcd.print("30010");
    lcd.setCursor(11,1);
    lcd.print(regBank.get(30010));
    Dcount=1;
 }
 break;

case '0':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("All Digital Inputs");
 lcd.setCursor(0,1);
 lcd.print("Set To False");

 //Sets all Digital Inputs to FALSE
 //Digital Inputs

 regBank.set(10001,0);
 regBank.set(10002,0);
 regBank.set(10003,0);
 regBank.set(10004,0);
 regBank.set(10005,0);
 regBank.set(10006,0);
 regBank.set(10007,0);
 regBank.set(10008,0);
 regBank.set(10009,0);
 regBank.set(10010,0);

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");

break;
```

```
case '1':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("All Digital Inputs");
 lcd.setCursor(0,1);
 lcd.print("Set To True");

 //Sets all coils to TRUE
 regBank.set(10001,1);
 regBank.set(10002,1);
 regBank.set(10003,1);
 regBank.set(10004,1);
 regBank.set(10005,1);
 regBank.set(10006,1);
 regBank.set(10007,1);
 regBank.set(10008,1);
 regBank.set(10009,1);
 regBank.set(10010,1);

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");

break;
```

```
case '2':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Analog Inputs");
 lcd.setCursor(0,1);
 lcd.print("Set To Zero");

 //Sets all Holding Registers to Zero
 regBank.set(30001, (word) 0);
 delay(10);
 regBank.set(30002, (word) 0);
 delay(10);
 regBank.set(30003, (word) 0);
 delay(10);
 regBank.set(30004, (word) 0);
 delay(10);
 regBank.set(30005, (word) 0);
 delay(10);
 regBank.set(30006, (word) 0);
 delay(10);
 regBank.set(30007, (word) 0);
 delay(10);
 regBank.set(30008, (word) 0);
 delay(10);
 regBank.set(30009, (word) 0);
 delay(10);
 regBank.set(30010, (word) 0);
 delay(10);
 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");


break;

case '3':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Analog Inputs");
 lcd.setCursor(0,1);
 lcd.print("HR# * 10^3");

 //Test numbers for Analogue Input Registers
 regBank.set(30001, (word) 1000);
 delay(10);
 regBank.set(30002, (word) 2000);
 delay(10);
 regBank.set(30003, (word) 3000);
 delay(10);
 regBank.set(30004, (word) 4000);
 delay(10);
 regBank.set(30005, (word) 5000);
 delay(10);
 regBank.set(30006, (word) 6000);
 delay(10);
 regBank.set(30007, (word) 7000);
 delay(10);
 regBank.set(30008, (word) 8000);
 delay(10);
 regBank.set(30009, (word) 9000);
 delay(10);
 regBank.set(30010, (word) 10000);
 delay(10);

 delay(2000);
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("A=FC01 B=FC02");
 lcd.setCursor(0,1);
 lcd.print("C=FC03 D=FC04");


break;

case '5':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
 lcd.print("Set To 32767");
```

```
//Test numbers for Analogue Input Registers
regBank.set(30001, (word) 32767);
delay(10);
regBank.set(30002, (word) 32767);
delay(10);
regBank.set(30003, (word) 32767);
delay(10);
regBank.set(30004, (word) 32767);
delay(10);
regBank.set(30005, (word) 32767);
delay(10);
regBank.set(30006, (word) 32767);
delay(10);
regBank.set(30007, (word) 32767);
delay(10);
regBank.set(30008, (word) 32767);
delay(10);
regBank.set(30009, (word) 32767);
delay(10);
regBank.set(30010, (word) 32767);
delay(10);

delay(2000);
lcd.clear();
lcd.setCursor(0,0);
lcd.print("A=FC01 B=FC02");
lcd.setCursor(0,1);
```

```
 lcd.print("C=FC03 D=FC04");



break;

case '6':
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Holding Registers");
 lcd.setCursor(0,1);
 lcd.print("Set To 65535");

 //Test numbers for Analogue Input Registers
 regBank.set(30001, (word) 65535);
 delay(10);
 regBank.set(30002, (word) 65535);
 delay(10);
 regBank.set(30003, (word) 65535);
 delay(10);
 regBank.set(30004, (word) 65535);
 delay(10);
 regBank.set(30005, (word) 65535);
 delay(10);
 regBank.set(30006, (word) 65535);
 delay(10);
 regBank.set(30007, (word) 65535);
 delay(10);
 regBank.set(30008, (word) 65535);
```

```
        delay(10);
        regBank.set(30009, (word) 65535);
        delay(10);
        regBank.set(30010, (word) 65535);
        delay(10);

        delay(2000);
        lcd.clear();
        lcd.setCursor(0,0);
        lcd.print("A=FC01 B=FC02");
        lcd.setCursor(0,1);
        lcd.print("C=FC03 D=FC04");
       break;
   }
}
```

C.3 ModbusRtu.h Code

```
/**
 * @file         ModbusRtu.h
 * @version      1.21
 * @date         2016.02.21
 * @author       Samuel Marco i Armengol
 * @contact      sammarcoarmengol@gmail.com
 * @contribution Helium6072
 *
 * @description
 *  Arduino library for communicating with Modbus devices
 *  over RS232/USB/485 via RTU protocol.
```

```
*
*  Further information:
*  http://modbus.org/
*  http://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf
*
* @license
*  This library is free software; you can redistribute it and/or
*  modify it under the terms of the GNU Lesser General Public
*  License as published by the Free Software Foundation; version
*  2.1 of the License.
*
*  This library is distributed in the hope that it will be useful,
*  but WITHOUT ANY WARRANTY; without even the implied warranty of
*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
*  Lesser General Public License for more details.
*
*  You should have received a copy of the GNU Lesser General Public
*  License along with this library; if not, write to the Free Software
*  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*
* @defgroup setup Modbus Object Instantiation/Initialization
* @defgroup loop Modbus Object Management
* @defgroup buffer Modbus Buffer Management
* @defgroup discrete Modbus Function Codes for Discrete Coils/Inputs
* @defgroup register Modbus Function Codes for Holding/Input Registers
*
*/
```

```
#include <inttypes.h>
#include "Arduino.h"
#include "Print.h"
#include <SoftwareSerial.h>

/**
 * @struct modbus_t
 * @brief
 * Master query structure:
 * This includes all the necessary fields to make the Master generate a Modbus query.
 * A Master may keep several of these structures and send them cyclically or
 * use them according to program needs.
 */
typedef struct
{
   uint8_t u8id;          /*!< Slave address between 1 and 247. 0 means broadcast */
   uint8_t u8fct;         /*!< Function code: 1, 2, 3, 4, 5, 6, 15 or 16 */
   uint16_t u16RegAdd;    /*!< Address of the first register to access at slave/s */
   uint16_t u16CoilsNo;   /*!< Number of coils or registers to access */
   uint16_t *au16reg;     /*!< Pointer to memory image in master */
}
modbus_t;

enum
{
   RESPONSE_SIZE = 6,
   EXCEPTION_SIZE = 3,
   CHECKSUM_SIZE = 2
};

/**
 * @enum MESSAGE
 * @brief
 * Indexes to telegram frame positions
 */
enum MESSAGE
{
   ID                         = 0, //!< ID field
   FUNC, //!< Function code position
   ADD_HI, //!< Address high byte
   ADD_LO, //!< Address low byte
   NB_HI, //!< Number of coils or registers high byte
   NB_LO, //!< Number of coils or registers low byte
   BYTE_CNT  //!< byte counter
};

/**
 * @enum MB_FC
 * @brief
 * Modbus function codes summary.
 * These are the implement function codes either for Master or for Slave.
 *
 * @see also fctsupported
 * @see also modbus_t
 */
enum MB_FC
```

```
{
   MB_FC_NONE                    = 0,   /*!< null operator */
   MB_FC_READ_COILS              = 1,       /*!< FCT=1 -> read coils or digital outputs */
   MB_FC_READ_DISCRETE_INPUT      = 2,       /*!< FCT=2 -> read digital inputs */
   MB_FC_READ_REGISTERS          = 3,  /*!< FCT=3 -> read registers or analog outputs */
   MB_FC_READ_INPUT_REGISTER      = 4,       /*!< FCT=4 -> read analog inputs */
   MB_FC_WRITE_COIL              = 5,       /*!< FCT=5 -> write single coil or output */
   MB_FC_WRITE_REGISTER          = 6,  /*!< FCT=6 -> write single register */
   MB_FC_WRITE_MULTIPLE_COILS     = 15,    /*!< FCT=15 -> write multiple coils or
outputs */
   MB_FC_WRITE_MULTIPLE_REGISTERS = 16          /*!< FCT=16 -> write multiple
registers */
};

enum COM_STATES
{
   COM_IDLE                   = 0,
   COM_WAITING                = 1


};

enum ERR_LIST
{
   ERR_NOT_MASTER              = -1,
   ERR_POLLING                 = -2,
   ERR_BUFF_OVERFLOW           = -3,
   ERR_BAD_CRC                 = -4,
   ERR_EXCEPTION               = -5
};
```

```
enum
{
   NO_REPLY = 255,
   EXC_FUNC_CODE = 1,
   EXC_ADDR_RANGE = 2,
   EXC_REGS_QUANT = 3,
   EXC_EXECUTE = 4
};

const unsigned char fctsupported[] =
{
   MB_FC_READ_COILS,
   MB_FC_READ_DISCRETE_INPUT,
   MB_FC_READ_REGISTERS,
   MB_FC_READ_INPUT_REGISTER,
   MB_FC_WRITE_COIL,
   MB_FC_WRITE_REGISTER,
   MB_FC_WRITE_MULTIPLE_COILS,
   MB_FC_WRITE_MULTIPLE_REGISTERS
};

#define T35  5
#define  MAX_BUFFER  64  //!< maximum size for the communication buffer in bytes

/**
 * @class Modbus
 * @brief
 * Arduino class library for communicating with Modbus devices over
 * USB/RS232/485 (via RTU protocol).
 */
class Modbus
{
private:
    HardwareSerial *port; //!< Pointer to Serial class object
    SoftwareSerial *softPort; //!< Pointer to SoftwareSerial class object
    uint8_t u8id; //!< 0=master, 1..247=slave number
    uint8_t u8serno; //!< serial port: 0-Serial, 1..3-Serial1..Serial3; 4: use software serial
    uint8_t u8txenpin; //!< flow control pin: 0=USB or RS-232 mode, >0=RS-485 mode
    uint8_t u8state;
    uint8_t u8lastError;
    uint8_t au8Buffer[MAX_BUFFER];
    uint8_t u8BufferSize;
    uint8_t u8lastRec;
    uint16_t *au16regs;
    uint16_t u16InCnt, u16OutCnt, u16errCnt;
    uint16_t u16timeOut;
    uint32_t u32time, u32timeOut;
    uint8_t u8regsize;

    void init(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin);
            void init(uint8_t u8id);
    void sendTxBuffer();
    int8_t getRxBuffer();
    uint16_t calcCRC(uint8_t u8length);
    uint8_t validateAnswer();
    uint8_t validateRequest();
    void get_FC1();
    void get_FC3();
    int8_t process_FC1( uint16_t *regs, uint8_t u8size );
    int8_t process_FC3( uint16_t *regs, uint8_t u8size );
    int8_t process_FC5( uint16_t *regs, uint8_t u8size );
    int8_t process_FC6( uint16_t *regs, uint8_t u8size );
    int8_t process_FC15( uint16_t *regs, uint8_t u8size );
    int8_t process_FC16( uint16_t *regs, uint8_t u8size );
    void buildException( uint8_t u8exception ); // build exception message

public:
    Modbus();
    Modbus(uint8_t u8id, uint8_t u8serno);
    Modbus(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin);
    Modbus(uint8_t u8id);
    void begin(long u32speed);
    void begin(SoftwareSerial *sPort, long u32speed);
    void begin(long u32speed, uint8_t u8config);
    void begin();
    void setTimeOut( uint16_t u16timeout); //!<write communication watch-dog timer
    uint16_t getTimeOut(); //!<get communication watch-dog timer value
    boolean getTimeOutState(); //!<get communication watch-dog timer state
    int8_t query( modbus_t telegram ); //!<only for master
    int8_t poll(); //!<cyclic poll for master
    int8_t poll( uint16_t *regs, uint8_t u8size ); //!<cyclic poll for slave
    uint16_t getInCnt(); //!<number of incoming messages
    uint16_t getOutCnt(); //!<number of outcoming messages
    uint16_t getErrCnt(); //!<error counter
    uint8_t getID(); //!<get slave ID between 1 and 247
    uint8_t getState();
    uint8_t getLastError(); //!<get last error message
    void setID( uint8_t u8id ); //!<write new ID for the slave
    void end(); //!<finish any communication and release serial communication port
};

/* _____PUBLIC
FUNCTIONS________________________________________________________ */

/**
 * @brief
 * Default Constructor for Master through Serial
 *
 * @ingroup setup
 */
Modbus::Modbus()
{
    init(0, 0, 0);
}

/**
 * @brief
 * Full constructor for a Master/Slave through USB/RS232C
 *
 * @param u8id   node address 0=master, 1..247=slave
 * @param u8serno  serial port used 0..3
 * @ingroup setup
 * @overload Modbus::Modbus(uint8_t u8id, uint8_t u8serno)
 * @overload Modbus::Modbus(uint8_t u8id)
 * @overload Modbus::Modbus()
 */
Modbus::Modbus(uint8_t u8id, uint8_t u8serno)
{
   init(u8id, u8serno, 0);
}

/**
 * @brief
 * Full constructor for a Master/Slave through USB/RS232C/RS485
 * It needs a pin for flow control only for RS485 mode
 *
 * @param u8id   node address 0=master, 1..247=slave
 * @param u8serno  serial port used 0..3
 * @param u8txenpin pin for txen RS-485 (=0 means USB/RS232C mode)
 * @ingroup setup
 * @overload Modbus::Modbus(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin)
 * @overload Modbus::Modbus(uint8_t u8id)
 * @overload Modbus::Modbus()
 */
Modbus::Modbus(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin)
{
   init(u8id, u8serno, u8txenpin);
}

/**
```

```
 * @brief
 * Constructor for a Master/Slave through USB/RS232C via software serial
 * This constructor only specifies u8id (node address) and should be only
 * used if you want to use software serial instead of hardware serial.
 * If you use this constructor you have to begin ModBus object by
 * using "void Modbus::begin(SoftwareSerial *softPort, long u32speed)".
 *
 * @param u8id   node address 0=master, 1..247=slave
 * @ingroup setup
 * @overload Modbus::Modbus(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin)
 * @overload Modbus::Modbus(uint8_t u8id, uint8_t u8serno)
 * @overload Modbus::Modbus()
 */
Modbus::Modbus(uint8_t u8id)
{
   init(u8id);
}

/**
 * @brief
 * Initialize class object.
 *
 * Sets up the serial port using specified baud rate.
 * Call once class has been instantiated, typically within setup().
 *
 * @see http://arduino.cc/en/Serial/Begin#.Uy4CJ6aKlHY
 * @param speed   baud rate, in standard increments (300..115200)
 * @ingroup setup
 */
void Modbus::begin(long u32speed)
{

   switch( u8serno )
   {
#if defined(UBRR1H)
   case 1:
       port = &Serial1;
       break;
#endif

#if defined(UBRR2H)
   case 2:
       port = &Serial2;
       break;
#endif

#if defined(UBRR3H)
   case 3:
       port = &Serial3;
       break;
#endif
   case 0:
   default:
       port = &Serial;
       break;
   }
```

```
  port->begin(u32speed);
  if (u8txenpin > 1)   // pin 0 & pin 1 are reserved for RX/TX
  {
    // return RS485 transceiver to transmit mode
    pinMode(u8txenpin, OUTPUT);
    digitalWrite(u8txenpin, LOW);
  }

  while(port->read() >= 0);
  u8lastRec = u8BufferSize = 0;
  u16InCnt = u16OutCnt = u16errCnt = 0;
}

/**
 * @brief
 * Initialize class object.
 *
 * Sets up the software serial port using specified baud rate and SoftwareSerial object.
 * Call once class has been instantiated, typically within setup().
 *
 * @param speed   *softPort, pointer to SoftwareSerial class object
 * @param speed   baud rate, in standard increments (300..115200)
 * @ingroup setup
 */
void Modbus::begin(SoftwareSerial *sPort, long u32speed)
{
    softPort=sPort;

    softPort->begin(u32speed);

    if (u8txenpin > 1)   // pin 0 & pin 1 are reserved for RX/TX
    {
        // return RS485 transceiver to transmit mode
        pinMode(u8txenpin, OUTPUT);
        digitalWrite(u8txenpin, LOW);
    }

    while(softPort->read() >= 0);
    u8lastRec = u8BufferSize = 0;
    u16InCnt = u16OutCnt = u16errCnt = 0;
}

/**
 * @brief
 * Initialize class object.
 *
 * Sets up the serial port using specified baud rate.
 * Call once class has been instantiated, typically within setup().
 *
 * @see http://arduino.cc/en/Serial/Begin#.Uy4CJ6aKlHY
 * @param speed   baud rate, in standard increments (300..115200)
 * @param config  data frame settings (data length, parity and stop bits)
 * @ingroup setup
 */
```

```
void Modbus::begin(long u32speed,uint8_t u8config)
{

  switch( u8serno )
  {
#if defined(UBRR1H)
  case 1:
    port = &Serial1;
    break;
#endif

#if defined(UBRR2H)
  case 2:
    port = &Serial2;
    break;
#endif

#if defined(UBRR3H)
  case 3:
    port = &Serial3;
    break;
#endif
  case 0:
  default:
    port = &Serial;
    break;
  }
```

```
    port->begin(u32speed, u8config);
    if (u8txenpin > 1)   // pin 0 & pin 1 are reserved for RX/TX
    {
        // return RS485 transceiver to transmit mode
        pinMode(u8txenpin, OUTPUT);
        digitalWrite(u8txenpin, LOW);
    }

    while(port->read() >= 0);
    u8lastRec = u8BufferSize = 0;
    u16InCnt = u16OutCnt = u16errCnt = 0;
}

/**
 * @brief
 * Initialize default class object.
 *
 * Sets up the serial port using 19200 baud.
 * Call once class has been instantiated, typically within setup().
 *
 * @overload Modbus::begin(uint16_t u16BaudRate)
 * @ingroup setup
 */
void Modbus::begin()
{
    begin(19200);
}
```

```
/**
 * @brief
 * Method to write a new slave ID address
 *
 * @param      u8id     new slave address between 1 and 247
 * @ingroup setup
 */
void Modbus::setID( uint8_t u8id)
{
   if (( u8id != 0) && (u8id <= 247))
   {
      this->u8id = u8id;
   }
}

/**
 * @brief
 * Method to read current slave ID address
 *
 * @return u8id          current slave address between 1 and 247
 * @ingroup setup
 */
uint8_t Modbus::getID()
{
   return this->u8id;
}

/**
```

```
 * @brief
 * Initialize time-out parameter
 *
 * Call once class has been instantiated, typically within setup().
 * The time-out timer is reset each time that there is a successful communication
 * between Master and Slave. It works for both.
 *
 * @param time-out value (ms)
 * @ingroup setup
 */
void Modbus::setTimeOut( uint16_t u16timeOut)
{
   this->u16timeOut = u16timeOut;
}

/**
 * @brief
 * Return communication Watchdog state.
 * It could be usefull to reset outputs if the watchdog is fired.
 *
 * @return TRUE if millis() > u32timeOut
 * @ingroup loop
 */
boolean Modbus::getTimeOutState()
{
   return (millis() > u32timeOut);
}
```

```
/**
 * @brief
 * Get input messages counter value
 * This can be useful to diagnose communication
 *
 * @return input messages counter
 * @ingroup buffer
 */
uint16_t Modbus::getInCnt()
{
   return u16InCnt;
}

/**
 * @brief
 * Get transmitted messages counter value
 * This can be useful to diagnose communication
 *
 * @return transmitted messages counter
 * @ingroup buffer
 */
uint16_t Modbus::getOutCnt()
{
   return u16OutCnt;
}

/**
 * @brief
 * Get errors counter value
 * This can be useful to diagnose communication
 *
 * @return errors counter
 * @ingroup buffer
 */
uint16_t Modbus::getErrCnt()
{
   return u16errCnt;
}

/**
 * Get modbus master state
 *
 * @return = 0 IDLE, = 1 WAITING FOR ANSWER
 * @ingroup buffer
 */
uint8_t Modbus::getState()
{
   return u8state;
}

/**
 * Get the last error in the protocol processor
 *
 * @returnreturn   NO_REPLY = 255      Time-out
 * @return   EXC_FUNC_CODE = 1   Function code not available
 * @return   EXC_ADDR_RANGE = 2  Address beyond available space for Modbus registers
 * @return   EXC_REGS_QUANT = 3  Coils or registers number beyond the available space
 * @ingroup buffer
 */
uint8_t Modbus::getLastError()
{
   return u8lastError;
}

/**
 * @brief
 * *** Only Modbus Master ***
 * Generate a query to an slave with a modbus_t telegram structure
 * The Master must be in COM_IDLE mode. After it, its state would be COM_WAITING.
 * This method has to be called only in loop() section.
 *
 * @see modbus_t
 * @param modbus_t  modbus telegram structure (id, fct, ...)
 * @ingroup loop
 * @todo finish function 15
 */
int8_t Modbus::query( modbus_t telegram )
{
   uint8_t u8regsno, u8bytesno;
   if (u8id!=0) return -2;
   if (u8state != COM_IDLE) return -1;

   if ((telegram.u8id==0) || (telegram.u8id>247)) return -3;
```

```
au16regs = telegram.au16reg;

// telegram header
au8Buffer[ ID ]         = telegram.u8id;
au8Buffer[ FUNC ]       = telegram.u8fct;
au8Buffer[ ADD_HI ]     = highByte(telegram.u16RegAdd );
au8Buffer[ ADD_LO ]     = lowByte( telegram.u16RegAdd );

switch( telegram.u8fct )
{
case MB_FC_READ_COILS:
case MB_FC_READ_DISCRETE_INPUT:
case MB_FC_READ_REGISTERS:
case MB_FC_READ_INPUT_REGISTER:
   au8Buffer[ NB_HI ]      = highByte(telegram.u16CoilsNo );
   au8Buffer[ NB_LO ]      = lowByte( telegram.u16CoilsNo );
   u8BufferSize = 6;
   break;
case MB_FC_WRITE_COIL:
   au8Buffer[ NB_HI ]      = ((au16regs[0] > 0) ? 0xff : 0);
   au8Buffer[ NB_LO ]      = 0;
   u8BufferSize = 6;
   break;
case MB_FC_WRITE_REGISTER:
   au8Buffer[ NB_HI ]      = highByte(au16regs[0]);
   au8Buffer[ NB_LO ]      = lowByte(au16regs[0]);
   u8BufferSize = 6;
   break;
```

```
case MB_FC_WRITE_MULTIPLE_COILS: // TODO: implement "sending coils"
   u8regsno = telegram.u16CoilsNo / 16;
   u8bytesno = u8regsno * 2;
   if ((telegram.u16CoilsNo % 16) != 0)
   {
      u8bytesno++;
      u8regsno++;
   }

   au8Buffer[ NB_HI ]      = highByte(telegram.u16CoilsNo );
   au8Buffer[ NB_LO ]      = lowByte( telegram.u16CoilsNo );
   au8Buffer[ NB_LO+1 ]    = u8bytesno;
   u8BufferSize = 7;

   u8regsno = u8bytesno = 0; // now auxiliary registers
   for (uint16_t i = 0; i < telegram.u16CoilsNo; i++)
   {


   }
   break;

case MB_FC_WRITE_MULTIPLE_REGISTERS:
   au8Buffer[ NB_HI ]      = highByte(telegram.u16CoilsNo );
   au8Buffer[ NB_LO ]      = lowByte( telegram.u16CoilsNo );
   au8Buffer[ NB_LO+1 ]    = (uint8_t) ( telegram.u16CoilsNo * 2 );
   u8BufferSize = 7;
```

```
        for (uint16_t i=0; i< telegram.u16CoilsNo; i++)
        {
            au8Buffer[ u8BufferSize ] = highByte( au16regs[ i ] );
            u8BufferSize++;
            au8Buffer[ u8BufferSize ] = lowByte( au16regs[ i ] );
            u8BufferSize++;
        }
        break;
    }

    sendTxBuffer();
    u8state = COM_WAITING;
    return 0;
}

/**
 * @brief *** Only for Modbus Master ***
 * This method checks if there is any incoming answer if pending.
 * If there is no answer, it would change Master state to COM_IDLE.
 * This method must be called only at loop section.
 * Avoid any delay() function.
 *
 * Any incoming data would be redirected to au16regs pointer,
 * as defined in its modbus_t query telegram.
 *
 * @params    nothing
 * @return errors counter
 * @ingroup loop
 */
int8_t Modbus::poll()
{
   // check if there is any incoming frame
         uint8_t u8current;
   if(u8serno<4)
      u8current = port->available();
   else
      u8current = softPort->available();

   if (millis() > u32timeOut)
   {
      u8state = COM_IDLE;
      u8lastError = NO_REPLY;
      u16errCnt++;
      return 0;
   }

   if (u8current == 0) return 0;

   // check T35 after frame end or still no frame end
   if (u8current != u8lastRec)
   {
      u8lastRec = u8current;
      u32time = millis() + T35;
      return 0;
   }
   if (millis() < u32time) return 0;
```

```
// transfer Serial buffer frame to auBuffer
u8lastRec = 0;
int8_t i8state = getRxBuffer();
if (i8state < 7)
{
   u8state = COM_IDLE;
   u16errCnt++;
   return i8state;
}

// validate message: id, CRC, FCT, exception
uint8_t u8exception = validateAnswer();
if (u8exception != 0)
{
   u8state = COM_IDLE;
   return u8exception;
}

// process answer
switch( au8Buffer[ FUNC ] )
{
case MB_FC_READ_COILS:
case MB_FC_READ_DISCRETE_INPUT:
   // call get_FC1 to transfer the incoming message to au16regs buffer
   get_FC1( );
   break;
case MB_FC_READ_INPUT_REGISTER:
    case MB_FC_READ_REGISTERS :
      // call get_FC3 to transfer the incoming message to au16regs buffer
      get_FC3( );
      break;
    case MB_FC_WRITE_COIL:
    case MB_FC_WRITE_REGISTER :
    case MB_FC_WRITE_MULTIPLE_COILS:
    case MB_FC_WRITE_MULTIPLE_REGISTERS :
      // nothing to do
      break;
    default:
      break;
    }
    u8state = COM_IDLE;
    return u8BufferSize;
}

/**
 * @brief
 * *** Only for Modbus Slave ***
 * This method checks if there is any incoming query
 * Afterwards, it would shoot a validation routine plus a register query
 * Avoid any delay() function !!!!
 * After a successful frame between the Master and the Slave, the time-out timer is reset.
 *
 * @param *regs  register table for communication exchange
 * @param u8size  size of the register table
 * @return 0 if no query, 1..4 if communication error, >4 if correct query processed
 * @ingroup loop
 */
int8_t Modbus::poll( uint16_t *regs, uint8_t u8size )
{

    au16regs = regs;
    u8regsize = u8size;
            uint8_t u8current;



    // check if there is any incoming frame
    if(u8serno<4)
        u8current = port->available();
    else
        u8current = softPort->available();

    if (u8current == 0) return 0;

    // check T35 after frame end or still no frame end
    if (u8current != u8lastRec)
    {
        u8lastRec = u8current;
        u32time = millis() + T35;
        return 0;
    }
    if (millis() < u32time) return 0;

    u8lastRec = 0;
```

```
int8_t i8state = getRxBuffer();
u8lastError = i8state;
if (i8state < 7) return i8state;

// check slave id
if (au8Buffer[ ID ] != u8id) return 0;

// validate message: CRC, FCT, address and size
uint8_t u8exception = validateRequest();
if (u8exception > 0)
{
  if (u8exception != NO_REPLY)
  {
    buildException( u8exception );
    sendTxBuffer();
  }
  u8lastError = u8exception;
  return u8exception;
}

u32timeOut = millis() + long(u16timeOut);
u8lastError = 0;

// process message
switch( au8Buffer[ FUNC ] )
{
case MB_FC_READ_COILS:
case MB_FC_READ_DISCRETE_INPUT:
        return process_FC1( regs, u8size );
        break;
    case MB_FC_READ_INPUT_REGISTER:
    case MB_FC_READ_REGISTERS :
        return process_FC3( regs, u8size );
        break;
    case MB_FC_WRITE_COIL:
        return process_FC5( regs, u8size );
        break;
    case MB_FC_WRITE_REGISTER :
        return process_FC6( regs, u8size );
        break;
    case MB_FC_WRITE_MULTIPLE_COILS:
        return process_FC15( regs, u8size );
        break;
    case MB_FC_WRITE_MULTIPLE_REGISTERS :
        return process_FC16( regs, u8size );
        break;
    default:
        break;
    }
    return i8state;
}

/* _____PRIVATE
FUNCTIONS____________________________________________________ */

void Modbus::init(uint8_t u8id, uint8_t u8serno, uint8_t u8txenpin)
{
    this->u8id = u8id;
    this->u8serno = (u8serno > 3) ? 0 : u8serno;
    this->u8txenpin = u8txenpin;
    this->u16timeOut = 1000;
}

void Modbus::init(uint8_t u8id)
{
    this->u8id = u8id;
    this->u8serno = 4;
    this->u8txenpin = 0;
    this->u16timeOut = 1000;
}

/**
 * @brief
 * This method moves Serial buffer data to the Modbus au8Buffer.
 *
 * @return buffer size if OK, ERR_BUFF_OVERFLOW if u8BufferSize >= MAX_BUFFER
 * @ingroup buffer
 */
int8_t Modbus::getRxBuffer()
{
    boolean bBuffOverflow = false;

    if (u8txenpin > 1) digitalWrite( u8txenpin, LOW );

    u8BufferSize = 0;
    if(u8serno<4)
        while ( port->available() )
        {
            au8Buffer[ u8BufferSize ] = port->read();
            u8BufferSize ++;

            if (u8BufferSize >= MAX_BUFFER) bBuffOverflow = true;
        }
    else
        while ( softPort->available() )
        {
            au8Buffer[ u8BufferSize ] = softPort->read();
            u8BufferSize ++;

            if (u8BufferSize >= MAX_BUFFER) bBuffOverflow = true;
        }
    u16InCnt++;

    if (bBuffOverflow)
    {
        u16errCnt++;
        return ERR_BUFF_OVERFLOW;
    }
    return u8BufferSize;
}

/**
 * @brief
 * This method transmits au8Buffer to Serial line.
 * Only if u8txenpin != 0, there is a flow handling in order to keep
 * the RS485 transceiver in output state as long as the message is being sent.
 * This is done with UCSRxA register.
 * The CRC is appended to the buffer before starting to send it.
 *
 * @param nothing
 * @return nothing
 * @ingroup buffer
 */
void Modbus::sendTxBuffer()
{
   uint8_t i = 0;

   // append CRC to message
   uint16_t u16crc = calcCRC( u8BufferSize );
   au8Buffer[ u8BufferSize ] = u16crc >> 8;
   u8BufferSize++;
   au8Buffer[ u8BufferSize ] = u16crc & 0x00ff;
   u8BufferSize++;

   // set RS485 transceiver to transmit mode
   if (u8txenpin > 1)
   {
      switch( u8serno )
      {
#if defined(UBRR1H)
      case 1:
          UCSR1A=UCSR1A |(1 << TXC1);
          break;
#endif

#if defined(UBRR2H)
        case 2:
          UCSR2A=UCSR2A |(1 << TXC2);
          break;
#endif

#if defined(UBRR3H)
        case 3:
          UCSR3A=UCSR3A |(1 << TXC3);
          break;
#endif
        case 0:
        default:
          UCSR0A=UCSR0A |(1 << TXC0);
          break;
        }
        digitalWrite( u8txenpin, HIGH );
    }

    // transfer buffer to serial line
    if(u8serno<4)
        port->write( au8Buffer, u8BufferSize );
    else
        softPort->write( au8Buffer, u8BufferSize );
```

```
   // keep RS485 transceiver in transmit mode as long as sending
   if (u8txenpin > 1)
   {
      switch( u8serno )
      {
#if defined(UBRR1H)
      case 1:
         while (!(UCSR1A & (1 << TXC1)));
         break;
#endif

#if defined(UBRR2H)
      case 2:
         while (!(UCSR2A & (1 << TXC2)));
         break;
#endif

#if defined(UBRR3H)
      case 3:
         while (!(UCSR3A & (1 << TXC3)));
         break;
#endif
      case 0:
      default:
         while (!(UCSR0A & (1 << TXC0)));
         break;
      }
```

```
        // return RS485 transceiver to receive mode
        digitalWrite( u8txenpin, LOW );
    }
    if(u8serno<4)
        while(port->read() >= 0);
    else
        while(softPort->read() >= 0);

    u8BufferSize = 0;

    // set time-out for master
    u32timeOut = millis() + (unsigned long) u16timeOut;

    // increase message counter
    u16OutCnt++;
}

/**
 * @brief
 * This method calculates CRC
 *
 * @return uint16_t calculated CRC value for the message
 * @ingroup buffer
 */
uint16_t Modbus::calcCRC(uint8_t u8length)
{
    unsigned int temp, temp2, flag;
    temp = 0xFFFF;
    for (unsigned char i = 0; i < u8length; i++)
    {
        temp = temp ^ au8Buffer[i];
        for (unsigned char j = 1; j <= 8; j++)
        {
            flag = temp & 0x0001;
            temp >>=1;
            if (flag)
                temp ^= 0xA001;
        }
    }
    // Reverse byte order.
    temp2 = temp >> 8;
    temp = (temp << 8) | temp2;
    temp &= 0xFFFF;
    // the returned value is already swapped
    // crcLo byte is first & crcHi byte is last
    return temp;
}

/**
 * @brief
 * This method validates slave incoming messages
 *
 * @return 0 if OK, EXCEPTION if anything fails
 * @ingroup buffer
 */
```

```
uint8_t Modbus::validateRequest()
{
   // check message crc vs calculated crc
   uint16_t u16MsgCRC =
      ((au8Buffer[u8BufferSize - 2] << 8)
       | au8Buffer[u8BufferSize - 1]); // combine the crc Low & High bytes
   if ( calcCRC( u8BufferSize-2 ) != u16MsgCRC )
   {
      u16errCnt ++;
      return NO_REPLY;
   }

   // check fct code
   boolean isSupported = false;
   for (uint8_t i = 0; i< sizeof( fctsupported ); i++)
   {
      if (fctsupported[i] == au8Buffer[FUNC])
      {
         isSupported = 1;
         break;
      }
   }
   if (!isSupported)
   {
      u16errCnt ++;
      return EXC_FUNC_CODE;
   }
```

```
// check start address & nb range
uint16_t u16regs = 0;
uint8_t u8regs;
switch ( au8Buffer[ FUNC ] )
{
case MB_FC_READ_COILS:
case MB_FC_READ_DISCRETE_INPUT:
case MB_FC_WRITE_MULTIPLE_COILS:
   u16regs = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ]) / 16;
   u16regs += word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ]) /16;
   u8regs = (uint8_t) u16regs;
   if (u8regs > u8regsize) return EXC_ADDR_RANGE;
   break;
case MB_FC_WRITE_COIL:
   u16regs = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ]) / 16;
   u8regs = (uint8_t) u16regs;
   if (u8regs > u8regsize) return EXC_ADDR_RANGE;
   break;
case MB_FC_WRITE_REGISTER :
   u16regs = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ]);
   u8regs = (uint8_t) u16regs;
   if (u8regs > u8regsize) return EXC_ADDR_RANGE;
   break;
case MB_FC_READ_REGISTERS :
case MB_FC_READ_INPUT_REGISTER :
case MB_FC_WRITE_MULTIPLE_REGISTERS :
   u16regs = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ]);
   u16regs += word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ]);
```

```
        u8regs = (uint8_t) u16regs;
        if (u8regs > u8regsize) return EXC_ADDR_RANGE;
        break;
    }
    return 0; // OK, no exception code thrown
}

/**
 * @brief
 * This method validates master incoming messages
 *
 * @return 0 if OK, EXCEPTION if anything fails
 * @ingroup buffer
 */
uint8_t Modbus::validateAnswer()
{
    // check message crc vs calculated crc
    uint16_t u16MsgCRC =
        ((au8Buffer[u8BufferSize - 2] << 8)
         | au8Buffer[u8BufferSize - 1]); // combine the crc Low & High bytes
    if ( calcCRC( u8BufferSize-2 ) != u16MsgCRC )
    {
        u16errCnt ++;
        return NO_REPLY;
    }

    // check exception
    if ((au8Buffer[ FUNC ] & 0x80) != 0)
    {
        u16errCnt ++;
        return ERR_EXCEPTION;
    }

    // check fct code
    boolean isSupported = false;
    for (uint8_t i = 0; i< sizeof( fctsupported ); i++)
    {
        if (fctsupported[i] == au8Buffer[FUNC])
        {
            isSupported = 1;
            break;
        }
    }
    if (!isSupported)
    {
        u16errCnt ++;
        return EXC_FUNC_CODE;
    }

    return 0; // OK, no exception code thrown
}

/**
 * @brief
 * This method builds an exception message
 *
 * @ingroup buffer
 */
void Modbus::buildException( uint8_t u8exception )
{
   uint8_t u8func = au8Buffer[ FUNC ];  // get the original FUNC code

   au8Buffer[ ID ]      = u8id;
   au8Buffer[ FUNC ]    = u8func + 0x80;
   au8Buffer[ 2 ]       = u8exception;
   u8BufferSize         = EXCEPTION_SIZE;
}

/**
 * This method processes functions 1 & 2 (for master)
 * This method puts the slave answer into master data buffer
 *
 * @ingroup register
 * TODO: finish its implementation
 */
void Modbus::get_FC1()
{
   uint8_t u8byte, i;
   u8byte = 0;

   //  for (i=0; i< au8Buffer[ 2 ] /2; i++) {
   //    au16regs[ i ] = word(
   //    au8Buffer[ u8byte ],
   //    au8Buffer[ u8byte +1 ]);
```

```
    //    u8byte += 2;
    //  }
}

/**
 * This method processes functions 3 & 4 (for master)
 * This method puts the slave answer into master data buffer
 *
 * @ingroup register
 */
void Modbus::get_FC3()
{
    uint8_t u8byte, i;
    u8byte = 3;

    for (i=0; i< au8Buffer[ 2 ] /2; i++)
    {
        au16regs[ i ] = word(
                        au8Buffer[ u8byte ],
                        au8Buffer[ u8byte +1 ]);
        u8byte += 2;
    }
}

/**
 * @brief
 * This method processes functions 1 & 2
 * This method reads a bit array and transfers it to the master
 *
 * @return u8BufferSize Response to master length
 * @ingroup discrete
 */
int8_t Modbus::process_FC1( uint16_t *regs, uint8_t u8size )
{
   uint8_t u8currentRegister, u8currentBit, u8bytesno, u8bitsno;
   uint8_t u8CopyBufferSize;
   uint16_t u16currentCoil, u16coil;

   // get the first and last coil from the message
   uint16_t u16StartCoil = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ] );
   uint16_t u16Coilno = word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ] );

   // put the number of bytes in the outcoming message
   u8bytesno = (uint8_t) (u16Coilno / 8);
   if (u16Coilno % 8 != 0) u8bytesno ++;
   au8Buffer[ ADD_HI ]  = u8bytesno;
   u8BufferSize         = ADD_LO;

   // read each coil from the register map and put its value inside the outcoming message
   u8bitsno = 0;

   for (u16currentCoil = 0; u16currentCoil < u16Coilno; u16currentCoil++)
   {
      u16coil = u16StartCoil + u16currentCoil;
      u8currentRegister = (uint8_t) (u16coil / 16);
      u8currentBit = (uint8_t) (u16coil % 16);
```

```
        bitWrite(
          au8Buffer[ u8BufferSize ],
          u8bitsno,
          bitRead( regs[ u8currentRegister ], u8currentBit ) );
        u8bitsno ++;

        if (u8bitsno > 7)
        {
          u8bitsno = 0;
          u8BufferSize++;
        }
    }

    // send outcoming message
    if (u16Coilno % 8 != 0) u8BufferSize ++;
    u8CopyBufferSize = u8BufferSize +2;
    sendTxBuffer();
    return u8CopyBufferSize;
}

/**
 * @brief
 * This method processes functions 3 & 4
 * This method reads a word array and transfers it to the master
 *
 * @return u8BufferSize Response to master length
 * @ingroup register
 */
int8_t Modbus::process_FC3( uint16_t *regs, uint8_t u8size )
{

   uint8_t u8StartAdd = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ] );
   uint8_t u8regsno = word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ] );
   uint8_t u8CopyBufferSize;
   uint8_t i;

   au8Buffer[ 2 ]       = u8regsno * 2;
   u8BufferSize         = 3;

   for (i = u8StartAdd; i < u8StartAdd + u8regsno; i++)
   {
      au8Buffer[ u8BufferSize ] = highByte(regs[i]);
      u8BufferSize++;
      au8Buffer[ u8BufferSize ] = lowByte(regs[i]);
      u8BufferSize++;
   }
   u8CopyBufferSize = u8BufferSize +2;
   sendTxBuffer();

   return u8CopyBufferSize;
}

/**
 * @brief
 * This method processes function 5
 * This method writes a value assigned by the master to a single bit
 *
 * @return u8BufferSize Response to master length
 * @ingroup discrete
 */
int8_t Modbus::process_FC5( uint16_t *regs, uint8_t u8size )
{
   uint8_t u8currentRegister, u8currentBit;
   uint8_t u8CopyBufferSize;
   uint16_t u16coil = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ] );

   // point to the register and its bit
   u8currentRegister = (uint8_t) (u16coil / 16);
   u8currentBit = (uint8_t) (u16coil % 16);

   // write to coil
   bitWrite(
      regs[ u8currentRegister ],
      u8currentBit,
      au8Buffer[ NB_HI ] == 0xff );



   // send answer to master
   u8BufferSize = 6;
   u8CopyBufferSize = u8BufferSize +2;
   sendTxBuffer();

   return u8CopyBufferSize;
}

/**
 * @brief
 * This method processes function 6
 * This method writes a value assigned by the master to a single word
 *
 * @return u8BufferSize Response to master length
 * @ingroup register
 */
int8_t Modbus::process_FC6( uint16_t *regs, uint8_t u8size )
{

    uint8_t u8add = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ] );
    uint8_t u8CopyBufferSize;
    uint16_t u16val = word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ] );

    regs[ u8add ] = u16val;

    // keep the same header
    u8BufferSize         = RESPONSE_SIZE;

    u8CopyBufferSize = u8BufferSize +2;
    sendTxBuffer();

    return u8CopyBufferSize;
}
```

```
/**
 * @brief
 * This method processes function 15
 * This method writes a bit array assigned by the master
 *
 * @return u8BufferSize Response to master length
 * @ingroup discrete
 */
int8_t Modbus::process_FC15( uint16_t *regs, uint8_t u8size )
{
   uint8_t u8currentRegister, u8currentBit, u8frameByte, u8bitsno;
   uint8_t u8CopyBufferSize;
   uint16_t u16currentCoil, u16coil;
   boolean bTemp;

   // get the first and last coil from the message
   uint16_t u16StartCoil = word( au8Buffer[ ADD_HI ], au8Buffer[ ADD_LO ] );
   uint16_t u16Coilno = word( au8Buffer[ NB_HI ], au8Buffer[ NB_LO ] );


   // read each coil from the register map and put its value inside the outcoming message
   u8bitsno = 0;
   u8frameByte = 7;
   for (u16currentCoil = 0; u16currentCoil < u16Coilno; u16currentCoil++)
   {

      u16coil = u16StartCoil + u16currentCoil;
      u8currentRegister = (uint8_t) (u16coil / 16);
```

```
        u8currentBit = (uint8_t) (u16coil % 16);

        bTemp = bitRead(
                    au8Buffer[ u8frameByte ],
                    u8bitsno );

        bitWrite(
            regs[ u8currentRegister ],
            u8currentBit,
            bTemp );

        u8bitsno ++;

        if (u8bitsno > 7)
        {
            u8bitsno = 0;
            u8frameByte++;
        }
    }

    // send outcoming message
    // it's just a copy of the incomping frame until 6th byte
    u8BufferSize         = 6;
    u8CopyBufferSize = u8BufferSize +2;
    sendTxBuffer();
    return u8CopyBufferSize;
}
```

```
/**
 * @brief
 * This method processes function 16
 * This method writes a word array assigned by the master
 *
 * @return u8BufferSize Response to master length
 * @ingroup register
 */
int8_t Modbus::process_FC16( uint16_t *regs, uint8_t u8size )
{
    uint8_t u8func = au8Buffer[ FUNC ];  // get the original FUNC code
    uint8_t u8StartAdd = au8Buffer[ ADD_HI ] << 8 | au8Buffer[ ADD_LO ];
    uint8_t u8regsno = au8Buffer[ NB_HI ] << 8 | au8Buffer[ NB_LO ];
    uint8_t u8CopyBufferSize;
    uint8_t i;
    uint16_t temp;

    // build header
    au8Buffer[ NB_HI ]   = 0;
    au8Buffer[ NB_LO ]   = u8regsno;
    u8BufferSize         = RESPONSE_SIZE;

    // write registers
    for (i = 0; i < u8regsno; i++)
    {
        temp = word(
                   au8Buffer[ (BYTE_CNT + 1) + i * 2 ],
                   au8Buffer[ (BYTE_CNT + 2) + i * 2 ]);
        regs[ u8StartAdd + i ] = temp;
    }
    u8CopyBufferSize = u8BufferSize +2;
    sendTxBuffer();

    return u8CopyBufferSize;
}
```

C.4 modbus.h Code

```
#include <stdint.h>
#ifndef _MODBUSPROTOCOL
#define _MODBUSPROTOCOL

//Maximum device list for network
#define DEVMAX              10
//Maximum control register que size
#define QUEMAX              10
//Maximum serial wait in micro seconds
#define SERIALMAXDELAY          100
```

```
#define SERIALBAUD                    9600
//the total silence time needed to signify an EOM or SOM in RTU mode


//Modbus function codes
#define READ_DO     0x01
#define READ_DI             0x02
#define READ_AO     0x03
#define READ_AI             0x04


#define WRITE_DO    0x05
#define WRITE_AO    0x06


#define RTU         0x01
#define ASCII       0x02


#define MASTER              0x01
#define SLAVE               0x02


#define DO                  0x00
#define DI                  0x01
#define AI                  0x03
#define AO                  0x04


#endif
```

C.5 modbusDevice.cpp Code

```
#include <modbusDevice.h>
```

```
modbusDevice::modbusDevice(void)
{
        _id=NULL;
}

void modbusDevice::setId(byte id)
{
        _id=id;
}

byte modbusDevice::getId(void)
{
        return(_id);
}
```

C.6 modbusDevice.h Code

```
#include <stdint.h>
#include <Arduino.h>

#ifndef _MODBUSDEVICE
#define _MODBUSDEVICE

#include <modbusRegBank.h>
#include <modbus.h>
//#include <Wprogram.h>

class modbusDevice:public modbusRegBank
```

```
{
        public:
                modbusDevice(void);
                void setId(byte id);
                byte getId(void);

        private:
                byte _id;
};
#endif
```

C.7 modbusRegbank.cpp Code

```
#include <modbusRegBank.h>
#include <stdlib.h>

modbusRegBank::modbusRegBank(void)
{
        _digRegs            = 0;
        _lastDigReg         = 0;
        _anaRegs            = 0;
        _lastAnaReg         = 0;
}


void modbusRegBank::add(word addr)
{
        if(addr<20000)
        {
```

```
        modbusDigReg *temp;

        temp = (modbusDigReg *) malloc(sizeof(modbusDigReg));
        temp->address = addr;
        temp->value         = 0;
        temp->next          = 0;

        if(_digRegs == 0)
        {
                _digRegs = temp;
                _lastDigReg = _digRegs;
        }
        else
        {
                //Assign the last register's next pointer to temp;
                _lastDigReg->next = temp;
                //then make temp the last register in the list.
                _lastDigReg = temp;
        }
}
else
{
        modbusAnaReg *temp;

        temp = (modbusAnaReg *) malloc(sizeof(modbusAnaReg));
        temp->address = addr;
        temp->value = 0;
        temp->next = 0;
```

```
		if(_anaRegs == 0)
		{
			_anaRegs = temp;
			_lastAnaReg = _anaRegs;
		}
		else
		{
			_lastAnaReg->next = temp;
			_lastAnaReg = temp;
		}
	}
}

word modbusRegBank::get(word addr)
{
	if(addr < 20000)
	{
		modbusDigReg * regPtr;
		regPtr = (modbusDigReg *) this->search(addr);
		if(regPtr)
			return(regPtr->value);
		else
			return(NULL);
	}
	else
	{
		modbusAnaReg * regPtr;
```

```
                regPtr = (modbusAnaReg *) this->search(addr);
                if(regPtr)
                        return(regPtr->value);
                else
                        return(NULL);
        }
}

void modbusRegBank::set(word addr, word value)
{
        //for digital data
        if(addr < 20000)
        {
                modbusDigReg * regPtr;
                //search for the register address
                regPtr = (modbusDigReg *) this->search(addr);
                //if a pointer was returned the set the register value to true if value is non zero
                if(regPtr)
                        if(value)
                                regPtr->value = 0xFF;
                        else
                                regPtr->value = 0x00;
        }
        else
        {
                modbusAnaReg * regPtr;
                //search for the register address
                regPtr = (modbusAnaReg *) this->search(addr);
```

```
		//if found then assign the register value to the new value.
		if(regPtr)
			regPtr->value = value;
	}
}

void * modbusRegBank::search(word addr)
{
	//if the requested address is 0-19999
	//use a digital register pointer assigned to the first digital register
	//else use a analog register pointer assigned the first analog register

	if(addr < 20000)
	{
		modbusDigReg *regPtr = _digRegs;

		//if there is no register configured, bail
		if(regPtr == 0)
			return(0);

		//scan through the linked list until the end of the list or the register is found.
		//return the pointer.
		do
		{
			if(regPtr->address == addr)
				return(regPtr);
			regPtr = regPtr->next;
		}
		while(regPtr);
	}
	else
	{
		modbusAnaReg *regPtr = _anaRegs;

		//if there is no register configured, bail
		if(regPtr == 0)
			return(0);

		//scan through the linked list until the end of the list or the register is found.
		//return the pointer.
		do
		{
			if(regPtr->address == addr)
				return(regPtr);
			regPtr = regPtr->next;
		}
		while(regPtr);
	}
	return(0);
}
```

C.8 modbusRegBank.h Code

```
#include <stdint.h>
#include <Arduino.h>
```

```
#ifndef _MODBUSREGBANK
#define _MODBUSREGBANK

#include <modbus.h>
//#include <Wprogram.h>


struct modbusDigReg
{
	word address;
	byte value;

	modbusDigReg *next;
};

struct modbusAnaReg
{
	word address;
	word value;

	modbusAnaReg *next;
};

class modbusRegBank
{
	public:

		modbusRegBank(void);
		void add(word);
		word get(word);
		void set(word, word);

	private:
		void * search(word);

		modbusDigReg		*_digRegs,
					*_lastDigReg;

		modbusAnaReg		*_anaRegs,
					*_lastAnaReg;
};
#endif
```

C.9 modbusSlave.cpp Code

```
#include <modbusSlave.h>
#include <modbus.h>
#include <modbusDevice.h>
#include <Arduino.h>

modbusSlave::modbusSlave()
{
}
/*
Set the Serial Baud rate.
Reconfigure the UART for 8 data bits, no parity, and 1 stop bit.
```

```
and flush the serial port.
*/
void modbusSlave::setBaud(word baud)
{
	_baud = baud;
	//calculate the time perdiod for 3 characters for the given bps in ms.
	_frameDelay = 24000/_baud;

	Serial.begin(baud);

	// defaults to 8-bit, no parity, 1 stop bit
	//clear parity, stop bits, word length
//	UCSR0C = UCSR0C & B11000001;
//	UCSR0B = UCSR0B & B11111011;

	//Set word length to 8 bits
//	UCSR0C = UCSR0C | B00000110;

	//No parity
//	UCSR0C = UCSR0C | B00000000;

	//1 Stop bit
//	UCSR0C = UCSR0C | B00000100;

	Serial.flush();
}

/*
```

```
Retrieve the serial baud rate
*/
word modbusSlave::getBaud(void)
{
	return(_baud);
}

/*
Generates the crc for the current message in the buffer.
*/

void modbusSlave::calcCrc(void)
{
	byte	CRCHi = 0xFF,
			CRCLo = 0x0FF,
			Index,
			msgLen,
			*msgPtr;

	msgLen = _len-2;
	msgPtr = _msg;

	while(msgLen--)
	{
		Index = CRCHi ^ *msgPtr++;
		CRCHi = CRCLo ^ _auchCRCHi[Index];
		CRCLo = _auchCRCLo[Index];
	}
```

```
          _crc = (CRCHi << 8) | CRCLo;
}

/*
  Checks the UART for query data
*/
void modbusSlave::checkSerial(void)
{
          //while there is more data in the UART than when last checked
          while(Serial.available()> _len)
          {
                    //update the incoming query message length
                    _len = Serial.available();
                    //Wait for 3 bytewidths of data (SOM/EOM)
//                  delayMicroseconds(RTUFRAMETIME);
                    delay(_frameDelay);
                    //Check the UART again
          }
}

/*
Copies the contents of the UART to a buffer
*/
void modbusSlave::serialRx(void)
{
          byte i;

          //allocate memory for the incoming query message
```

```
	_msg = (byte*) malloc(_len);

		//copy the query byte for byte to the new buffer
		for (i=0 ; i < _len ; i++)
			_msg[i] = Serial.read();
}

/*
Generates a query reply message for Digital In/Out status update queries.
*/
void modbusSlave::getDigitalStatus(byte funcType, word startreg, word numregs)
{
	//initialize the bit counter to 0
	byte bitn =0;

	//if the function is to read digital inputs then add 10001 to the start register
	//else add 1 to the start register
	if(funcType == READ_DI)
		startreg += 10001;
	else
		startreg += 1;

	//determine the message length
	//for each group of 8 registers the message length increases by 1
	_len = numregs/8;
	//if there is at least one incomplete byte's worth of data
	//then add 1 to the message length for the partial byte.
	if(numregs%8)
```

```
        _len++;
    //allow room for the Device ID byte, Function type byte, data byte count byte, and crc
word
    _len +=5;


    //allocate memory of the appropriate size for the message
    _msg = (byte *) malloc(_len);


    //write the slave device ID
    _msg[0] = _device->getId();
    //write the function type
    _msg[1] = funcType;
    //set the data byte count
    _msg[2] = _len-5;


    //For the quantity of registers queried
    while(numregs--)
    {
        //if a value is found for the current register, set bit number bitn of msg[3]
        //else clear it
        if(_device->get(startreg))
            bitSet(_msg[3], bitn);
        else
            bitClear(_msg[3], bitn);
        //increment the bit index
        bitn++;
        //increment the register
        startreg++;
    }
```

```
	//generate the crc for the query reply and append it
	this->calcCrc();
	_msg[_len - 2] = _crc >> 8;
	_msg[_len - 1] = _crc & 0xFF;
}

void modbusSlave::getAnalogStatus(byte funcType, word startreg, word numregs)
{
	word val;
	word i = 0;

	//if the function is to read analog inputs then add 30001 to the start register
	//else add 40001 to the start register
	if(funcType == READ_AI)
		startreg += 30001;
	else
		startreg += 40001;

	//calculate the query reply message length
	//for each register queried add 2 bytes
	_len = numregs * 2;
	//allow room for the Device ID byte, Function type byte, data byte count byte, and crc
word
	_len += 5;

	//allocate memory for the query response
	_msg = (byte *) malloc(_len);
```

```
        //write the device ID
        _msg[0] = _device->getId();
        //write the function type
        _msg[1] = funcType;
        //set the data byte count
        _msg[2] = _len - 5;

        //for each register queried
        while(numregs--)
        {
                //retrieve the value from the register bank for the current register
                val = _device->get(startreg+i);
                //write the high byte of the register value
                _msg[3 + i * 2]  = val >> 8;
                //write the low byte of the register value
                _msg[4 + i * 2] = val & 0xFF;
                //increment the register
                i++;
        }

        //generate the crc for the query reply and append it
        this->calcCrc();
        _msg[_len - 2] = _crc >> 8;
        _msg[_len - 1] = _crc & 0xFF;
}

void modbusSlave::setStatus(byte funcType, word reg, word val)
{
```

```
//Set the query response message length
//Device ID byte, Function byte, Register byte, Value byte, CRC word
_len = 8;
//allocate memory for the message buffer.
_msg = (byte *) malloc(_len);


//write the device ID
_msg[0] = _device->getId();
//if the function type is a digital write
if(funcType == WRITE_DO)
{
        //Add 1 to the register value and set it's value to val
        _device->set(reg + 1, val);
        //write the function type to the response message
        _msg[1] = WRITE_DO;
}
else
{
        //else add 40001 to the register and set it's value to val
        _device->set(reg + 40001, val);

        //write the function type of the response message
        _msg[1] = WRITE_AO;
}

//write the register number high byte value
_msg[2] = reg >> 8;
```

```
        //write the register number low byte value
        _msg[3] = reg & 0xFF;
        //write the control value's high byte
        _msg[4] = val >> 8;
        //write the control value's low byte
        _msg[5] = val & 0xFF;

        //calculate the crc for the query reply and append it.
        this->calcCrc();
        _msg[_len - 2]= _crc >> 8;
        _msg[_len - 1]= _crc & 0xFF;
}

void modbusSlave::run(void)
{

        byte deviceId;
        byte funcType;
        word field1;
        word field2;

        int i;

        //initialize mesasge length
        _len = 0;

        //check for data in the recieve buffer
        this->checkSerial();
```

```
//if there is nothing in the recieve buffer, bail.
if(_len == 0)
{
	return;
}

//retrieve the query message from the serial uart
this->serialRx();

//if the message id is not 255, and
// and device id does not match bail
if( (_msg[0] != 0xFF) &&
	(_msg[0] != _device->getId()) )
{
	return;
}
//calculate the checksum of the query message minus the checksum it came with.
this->calcCrc();

//if the checksum does not match, ignore the message
if ( _crc != ((_msg[_len - 2] << 8) + _msg[_len - 1]))
	return;

//copy the function type from the incoming query
funcType = _msg[1];

//copy field 1 from the incoming query
```

```
field1   = (_msg[2] << 8) | _msg[3];

//copy field 2 from the incoming query
field2  = (_msg[4] << 8) | _msg[5];

//free the allocated memory for the query message
free(_msg);
//reset the message length;
_len = 0;

//generate query response based on function type
switch(funcType)
{
case READ_DI:
        this->getDigitalStatus(funcType, field1, field2);
        break;
case READ_DO:
        this->getDigitalStatus(funcType, field1, field2);
        break;
case READ_AI:
        this->getAnalogStatus(funcType, field1, field2);
        break;
case READ_AO:
        this->getAnalogStatus(funcType, field1, field2);
        break;
case WRITE_DO:
        this->setStatus(funcType, field1, field2);
        break;
        case WRITE_AO:
                this->setStatus(funcType, field1, field2);
                break;
        default:
                return;
                break;
        }

        //if a reply was generated
        if(_len)
        {
                int i;
                //send the reply to the serial UART
                //Senguino doesn't support a bulk serial write command....
                for(i = 0 ; i < _len ; i++)
                        Serial.write(_msg[i]);
                //free the allocated memory for the reply message
                free(_msg);
                //reset the message length
                _len = 0;
        }
}
```

C.10 modbusSlave.h Code

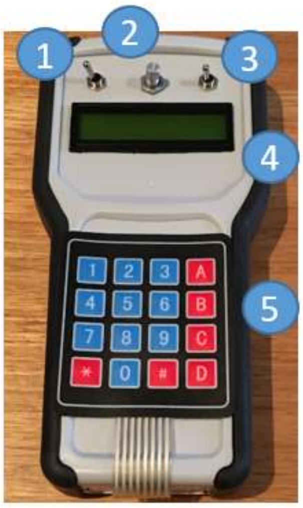
```
#include <stdint.h>
#include <Arduino.h>

#ifndef MODBUSSLAVE
```

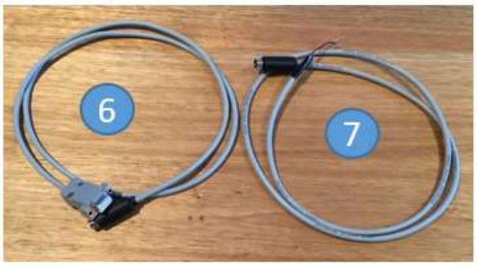
```
#define MODBUSSLAVE

#include <modbus.h>
#include <modbusDevice.h>
//#include <Wprogram.h>

/* Table of CRC values for high–order byte */
const byte _auchCRCHi[] = {
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40};

/* Table of CRC values for low–order byte */
```

```
const byte _auchCRCLo[] = {

0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,

0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9,
0x09,

0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F,
0xDD,

0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13,
0xD3,

0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,

0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA,
0x3A,

0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA,
0xEE,

0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6,
0x26,

0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,

0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF,
0x6F,

0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79,
0xBB,

0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75,
0xB5,

0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,

0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,

0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,

0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C,
0x8C,

0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,

0x40};


class modbusSlave

{
	public:
		modbusSlave(void);
		void setBaud(word);
		word getBaud(void);
		void calcCrc(void);
		void checkSerial(void);
		void serialRx(void);
		void getDigitalStatus(byte, word, word);
		void getAnalogStatus(byte, word, word);
		void setStatus(byte, word, word);
		void run(void);

		modbusDevice *_device;

	private:
		byte *_msg,
			_len;

		word _baud,
			_crc,
			_frameDelay;
};
#endif
```

Appendix D User Guide

The following is a simple guide in how to use the Arduino Modbus Simulator





1. RS-232 RS-485 Selector Switch
2. LCD screen contrast adjustment
3. Power On / Off switch
4. LCD display
5. Keypad
6. RS-232 cable
7. RS-485 cable
8. Communications port
9. Battery hatch

Power Up

Install the 9V battery by removing the hatch and connecting into the clip.

Power is turned on by moving power switch (3) to the left position.

Correct the screen contrast by adjusting the rotatable knob (2).

Communications Setup

Select communications layer with the Selector Switch (1). Left position is for RS-485 and right position is for RS-232.

The default communication settings 19200 Baud, 8 bits word length, no parity and 1 stop bit.

Adjustment to these parameters are only available via software update.

For a RS-232 network, use the RS-232 cable (6). For a RS-485, use the RS-485 cable (7) where the Orange core is Tx and Black core is Rx. Plug the appropriate cable into the mini din communications port (8).

Display

Upon power up the display will go to main menu.

Pressing ‘A’ will move to the Coil values beginning with address 00001

Continue to press A until the user has looped through the 10 Coils.

Pressing ‘*’ at any time will return user to the main menu.

Pressing ‘B’ at any time will move user to the Digital Input values beginning with address 10001

Continue to press ‘B’ until the user has looped through the 10 Digital Inputs.

Pressing ‘C’ at any time will move user to the Holding Register values beginning with address 40001

Continue to press ‘C’ until the user has looped through the 10 Holding Registers.

Pressing ‘D’ at any time will move user to the Analogue Input values beginning with address 30001

Continue to press ‘D’ until the user has looped through the 10 Analogue Input Registers.

Pressing ‘0’ will pre-set all Coils to a value of False (0). Pressing ‘1’ will pre-set all Coils to a value of True (1).

Pressing ‘2’ will pre-set all Holding Register values to zero.

Pressing ‘3 will set the Holding Register values to the following:

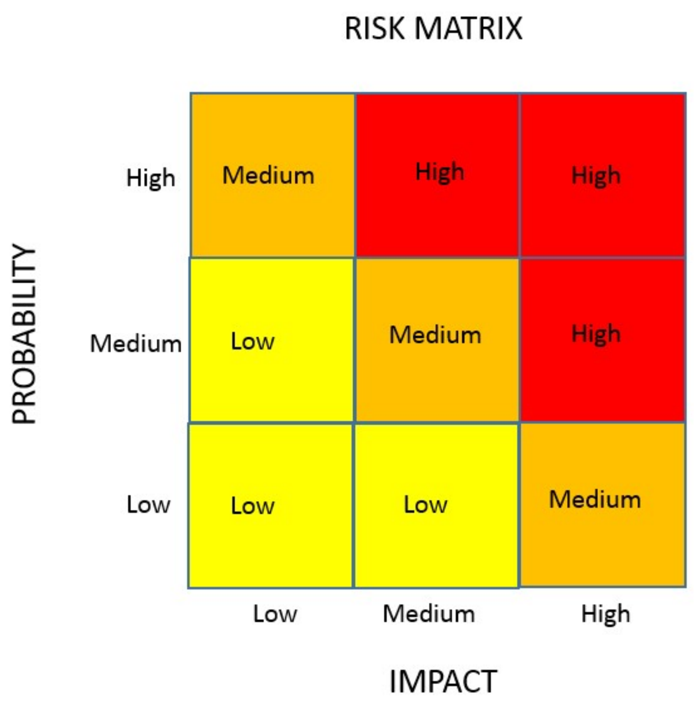
40001=0;	40002=1000;	40003=2000;	40004=3000;	40005=4000;
40006=5000;	40007=6000;	40008=7000;	40009=8000;	40010=9000;

Pressing ‘5’ will pre-set all Holding Register values to 32767.

Pressing ‘6’ will pre-set all Holding Register values to 65535.

Appendix E Risk Assessment

Using the below risk assessment matrix, the risks for both project execution and health & safety have been assessed and mitigated. 5 key health & safety risks and 3 key project risk have been identified.



Health & Safety Impact
Low Impact: Minor first aid
Medium Impact: Medical treatment
High Impact: Fatality or Permanent Disability

Project Execution Impact
Low Impact: Less than 2 days
Medium Impact: 2 to 7 days
High Impact: Greater than 7 days

Risk	Probability	Impact	Risk	Mitigation
Electric shock	Low	High	Medium	The electrical circuits for communications and Arduino are 3.3 to 5 volts DC. This is safe voltage levels
Toxic fumes from soldering	High	Medium	High	Use lead free and non-toxic solder
Burns during 3D printing of parts for enclosure	Low	Medium	Low	Ensure 3D printer door enclosure is closed when in use and use correct manual handling tools for handing prints
Eye injury during soldering and electronic assembly	High	High	High	Wear safety glasses at all times when conducting these activities

Back or neck strain from working long hours at bench during hardware development	High	Medium	High	Ensure ergonomic body positioning. Frequency breaks and stretching
Risk	**Probability**	**Impact**	**Risk**	**Mitigation**
No access to PLC or other Modbus device for testing	Low	High	Medium	Contact equipment suppliers and request they support student project by supplying required hardware
Project schedule impacted by semester 1 workload – assignments and exam preparations	High	High	High	Move project start date earlier. Implement study plan for semester 1 units into project plan to determine critical path
Unforeseen technical delays	Medium	High	High	Seek counsel from Supervisors Seek counsel from practicing engineers Seek counsel from peer students