COMMUNICATION BETWEEN VIRTUAL EMULATION SYSTEM AND PLC BY MODBUS/TCP PROTOCOL

A Thesis

Presented to

the Faculty of the College of Science and Technology

Morehead State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Huang (Benjamin) Huang

November 20, 2015

ProQuest Number: 10187743

ProQuest 10187743

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Accepted by the faculty of the College of Science and Technology, Morehead State University, in partial fulfillment of the requirements for the Master of Science degree.

_____
Director of Thesis

Master's Committee: _____, Chair

_____

_____

_____
Date

COMMUNICATION BETWEEN VIRTUAL EMULATION SYSTEM AND PLC BY MODBUS/TCP PROTOCOL

Huang (Benjamin) Huang
Morehead State University, 2015

Director of Thesis: _____

A virtual laboratory plays an important role for academic purposes. Building this kind of laboratory by assembling the hardware including PLC, sensor, conveyor, machine, or robot to an industry emulation automation environment is a big job and only a limited number of equipment can be involved. An emulated environment running on a computer can be used to solve this problem, but how can one also control the virtual devices and collect information from a virtual devices to a real PLC?

This research focuses on the communication between the virtual emulation system running on a computer and the PLC, so that, by using the program running on the PLC, one can control the virtual robot. This thesis describes the implementation of the Modbus/TCP protocol in PLC and the main functionalities for virtual robot control.

Accepted by: _____, Chair

_____

_____

ACKNOWLEDGEMENT

The author would like to express his gratitude to all the professors and students who have reviewed this thesis and gave constructive ideas to improve the quality of the thesis.

I would like to first of all express my appreciation to Dr. Ortega-Moody, my Thesis Director, for providing the opportunity to research and develop a software system for this thesis. As my undergraduate and professional background is linked to computer science, it was vital for me to use my knowledge and improve my knowledge through this research. His constructive advice given throughout the rough periods of the research encouraged me to complete the thesis in a timely manner.

Secondly, I would like to express my gratitude for Dr. Zargari and Dr. Grisé for going through this thesis and providing important suggestions to articulate the problems and solutions for the other readers.

My sincere thankfulness is also expressed to Mr. Sanchez-Alonso for his valuable advice on this thesis.

Table of Contents

1. CHAPTER ONE: INTRODUCTION

1.1 Background

Technological advances have resulted in a significance revolution of automation in industry. The related research and training are becoming increasingly important both strategically and economically. However, building a laboratory for academic purposes by assembling the required hardware, including PLC's, sensors, a conveyor, machines, and/or robot for an industry emulation automation environment is a big job and only a limited number of different equipment and peripheral devices can be involved in the infrastructure of the laboratory. It is difficult to build this kind of laboratory because a real laboratory involves all of the types of hardware. Furthermore, automation is an area that is constantly being updated as new technology is developed. Most of the new invented equipment is expensive and needs time to be ordered and delivered. It is not practical to change and re-setup the hardware in the laboratory frequently.

Robot simulation involves simulating the robot program and the program of instructions in the virtual world of a computer graphics program. Program simulation is typically accomplished by modeling the robot, tooling, and peripheral equipment in the software and then executing the program of instructions for the application. (Daniel E. Kandray, P.E, 2010)

For these reasons, the development of a virtual industrial laboratory to simulate the real one is proposed. With this development, researchers will be able to change the configuration of the laboratory to fit any realistic industry environment that they want. This new trend known as "Serious Games" has its main focus the field of specialized instruction and training. (Zyda, M., 2005)

By using the PLC's connection to the industrial virtual laboratory, students can practice programing complex automation sequences without necessarily being in a real industrial environment. This means they will be able to modify the parameters and the sequences of control conveniently. A virtual laboratory is particularly useful when some experiments involve equipment that might cause harm to human beings, or risk compromising the safety of the process. Moreover, all of the operations and their consequences can be visualized in real-time. Another objective of a virtual laboratory is to provide hands-on lab activities ~~to~~ that can enhance online courses. By using the communication technology used on the internet, the virtual industrial laboratory can be accessed and shared on a global scale. Thus, research and training will be free from restrictions of time and location.

1.2 Purpose

In a realistic industry environment, the engineer will use the PLC to control the combination of conveyors, sensors, and machines. The instruction will be sent from the PLC to the peripheral hardware or robot, and the data will be collected by hardware or robot and sent back to the PLC directly. Correspondingly, in the virtual emulation system, the PLC needs to communicate with the virtual hardware or robot running on the computer.

This research will focus on the communication between the simulation system and the PLC. The Modbus/TCP protocol will be used for communication between the PLC and the virtual equipment. In the virtual simulation system, the PLC will work as a client and the virtual simulation system running on the computer will work as a server. The client–server model of programming is a distributed application structure that communicate between the providers of a resource or service, called servers, and service requesters, called clients. Through the combination between the server and the client, the PLC can control

the virtual hardware and get data from it.

Taking a virtual robot as an example, the status of the virtual robot can be recorded in the memory. To control the robot, therefore, means to modify those data in the memory (Figure 1.1). For example, if we want to move the effector, we can change the angles of the robot's joints. If we want the robot to pick up the piece, we can set the value corresponding to the effector to pick up or drop the object.

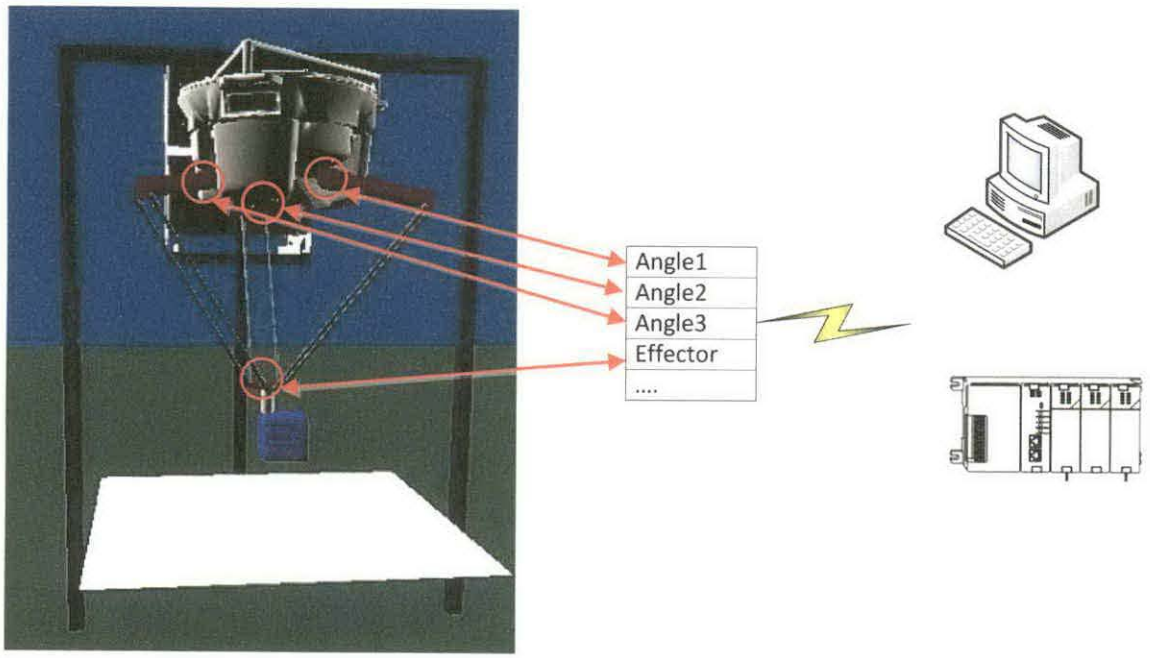


Figure 1.1 Data in Memory

Modbus is considered an application layer messaging protocol, providing Master/Slave communication (Figure 1.2) between devices connected together through buses or networks. By using it, the PLC or a computer can control the virtual robot remotely. Furthermore, the PLC can control multiple virtual devices running in the virtual emulation system.

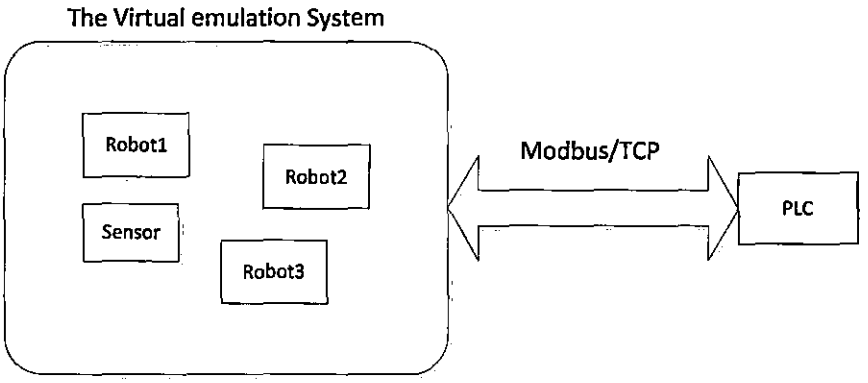


Figure 1.2 Master/Slave Communication

Because Modbus is supported by real devices, the benefit of using Modbus is the program running in the emulation system can later be immigrated to the real industry environment smoothly and conveniently (Figure 1.3 Modbus communication).

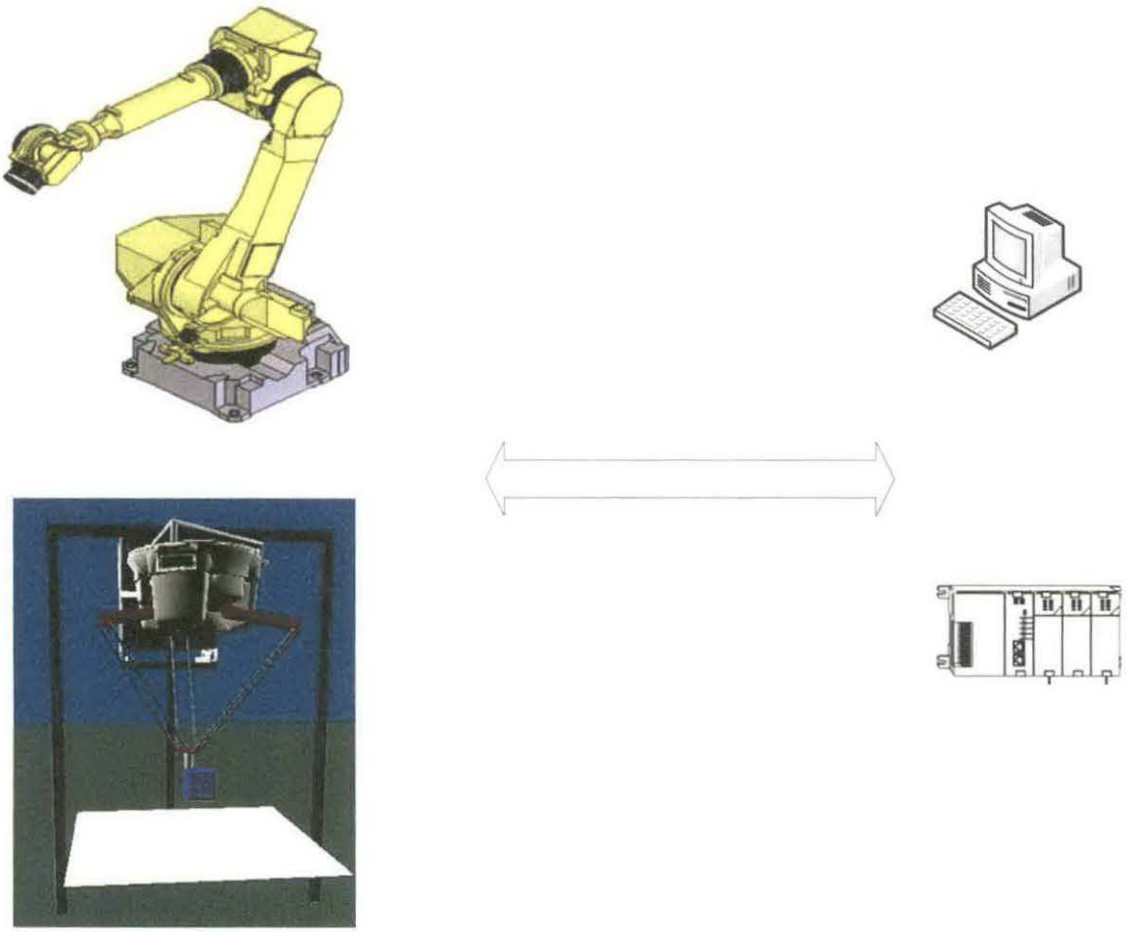


Figure 1.3 Modbus Communication

1.3 Research objectives

Objective 1:

Program the PLC in RSLogix 5000 and implement Modbus/TCP in the PLC as a client.

Objective 2:

Operate the virtual robot and monitor its state by reading and writing the virtual

coils and registers in the virtual robot.

Objective 3:

Make the virtual robot accomplish an action sequence, including moving to the target position, take object, move and put object at the target position.

1.4 Assumptions

The following assumptions are considered as pre-conditions before this research.

The Modbus/TCP server has been implemented in the virtual robot and can communicate with the client. It will listen at port 502 and be ready to accept connection from the client. After the connection was connected, the client can communicate with the virtual robot.

The Ethernet module is necessary for the PLC. The Ethernet module can connect the PLC to the internet or intranet and communicate with other devices connected to the network.

Currently, our research focuses on communication with an existing virtual robot (Figure 1.4). The virtual robot has the Modbus/TCP protocol implemented and can work as a server. The client program can operate it by reading and writing virtual coils and registers into it.

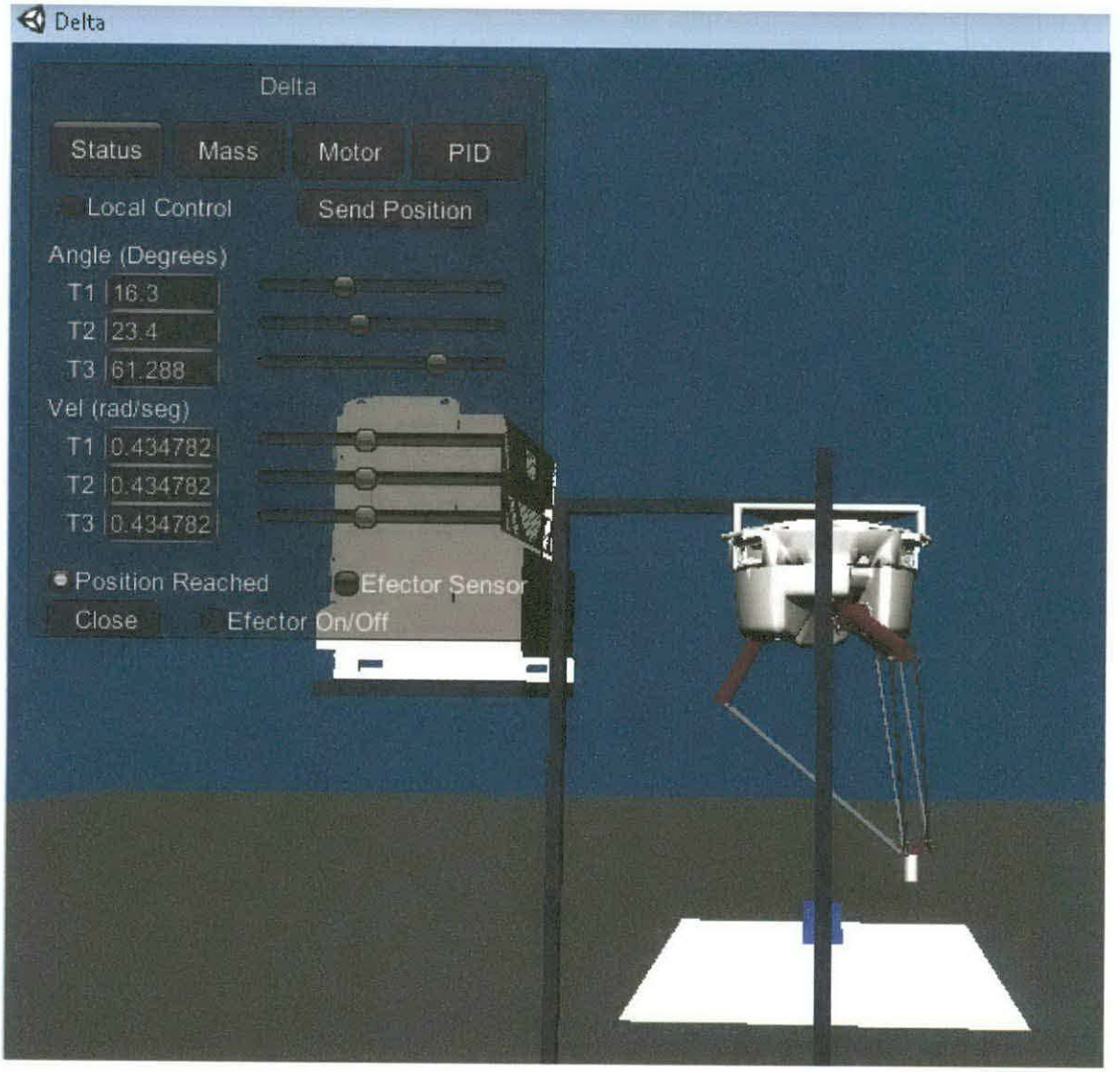


Figure 1.4 Virtual robot

1.5 Limitations

Not all PLCs have an Ethernet module. Only new PLCs with Ethernet modules can communicate with other devices through the Modbus/TCP protocol on Ethernet.

1.6 Definition of Terms

PLC:

"A programmable logic controller, PLC, is a user-friendly, microprocessor-based,

specialized computer that carries out control functions of many types and levels of complexity. It can be programmed, controlled and operated by a person unskilled in operating computers. The PLC's operator essentially draws the lines and devices of ladder diagrams" (John W. Webb, 1995).

Ethernet:

"Ethernet computer networking technologies works for local area networks (LANs) and metropolitan area networks (MANs). It is a wired network technology that is defined by IEEE8.2.3 standards. It was first deployed in 1976 and has since emerged as the dominant standard for wired connections in local area networks." (June Jamrich Parson, 2014) It has since been refined to support higher bit rates and longer link distances. The Ethernet RJ 45 (Figure 1.5) cable can be used to connect devices.

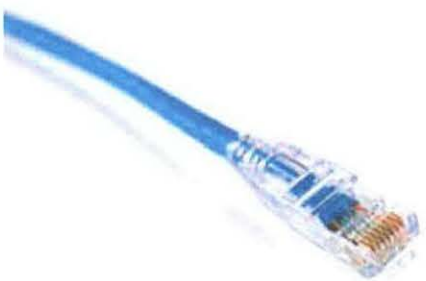


Figure 1.5 RJ-45

Sockets

Sockets are a concept developed at the University of California at Berkeley to add network communication support to the UNIX operating system. The API (Application Program Interface) developed there is now known as the "Berkeley socket interface."

Sockets are generally, but not exclusively, used in conjunction with the Transmission Control Protocol/Internet Protocol (TCP/IP) that dominates Internet communications. The

Internet Protocol (IP) part of TCP/IP involves packaging data into "datagrams" that contain header information to identify the source and destination of the data. The Transmission Control Protocol (TCP) provides a means of reliable transport and error checking for the IP datagrams.

Within TCP/IP, a communication endpoint is defined by an IP address and a port number. The IP address consists of 4 bytes that identify a server on the Internet. The IP address is generally shown in "dotted quad" format, with decimal numbers separated by periods, for example "209.86.105.231". "A port number identifies a particular service or process that the server provides. Some of these port numbers are standardized to provide well-known services". (Charles Petzold, 1998)

RSLogix 5000:

The RSLogix 5000 Enterprise Series software is designed to work with Logix5000 controller platforms. RSLogix 5000 Enterprise Series software is an IEC 61131-3 compliant software package that offers relay ladder, structured text, function block diagram, and sequential function chart editors for the designer to develop application programs. Create your own instructions by encapsulating a section of logic in any programming language into an Add-On Instruction. (RSLogix 5000 Programming Software)

Ladder Diagram:

Ladder diagrams are specialized schematics commonly used to document industrial control logic systems. They are called "ladder" diagrams because they resemble a ladder, with two vertical rails (supply power) and as many "rungs" (horizontal lines) as there are control circuits to represent. If we wanted to draw a simple ladder diagram showing a lamp that is controlled by a hand switch, it would look like this (Figure 1.6):

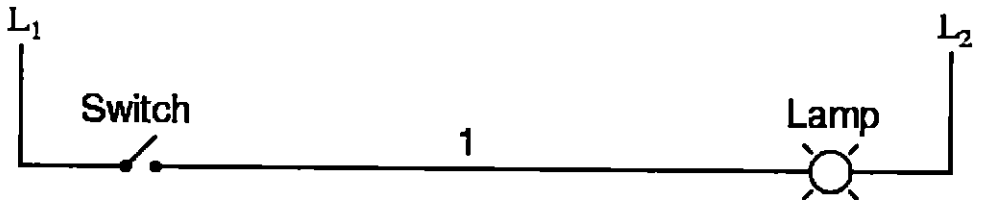


Figure 1.6 Ladder Diagrams

Ladder diagrams are used only on 2-wire control circuits, not for the power circuit of the driven or controlled equipment. (Kenneth G. Oliver, 1990.)

Modbus:

Modbus is a serial communication protocol initially developed by AEG-MOdicon. It was initially designed to operate with programmable logic controllers (PLCs). It is an application layer messaging protocol, operating at layer 7 of the Open Systems Interconnection (OSI) protocol, and provides client-server communication between devices connection on different types of net-work. (Sunit Kumar Sen, 2014)

Modbus/TCP:

Modbus TCP/IP (also Modbus-TCP) is simply the Modbus RTU protocol with a TCP interface that runs on Ethernet.

The Modbus messaging structure is the application protocol that defines the rules for organizing and interpreting the data independent of the data transmission medium.

TCP/IP refers to the Transmission Control Protocol and Internet Protocol, which provides the transmission medium for Modbus TCP/IP messaging. Simply stated, TCP/IP allows blocks of binary data to be exchanged between computers. It is also a world-wide standard that serves as the foundation for the World Wide Web. The primary function of TCP is to ensure that all packets of data are received correctly, while IP makes sure that

messages are correctly addressed and routed. Note that the TCP/IP combination is merely a transport protocol, and does not define what the data means or how the data is to be interpreted (this is the job of the application protocol, Modbus in this case). (Acromag, Inc.2005)

Crossover cable:

A crossover cable can directly connect two devices without a hub or switch. You can use a crossover cable to connect two computers directly to each other, but crossover cables are more often used to daisy-chain hubs and switches to each other.

If you want to create your own crossover cable, you must reverse the wires on one end of the cable, as shown in Figure1.6 (Doug Lowe, 2013)

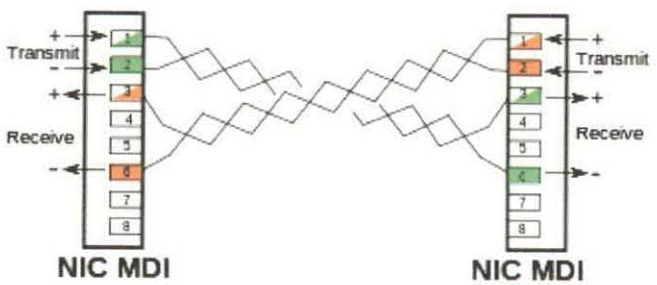


Figure1.6 Crossover cable

IP Address:

IP uses 32 bits, or four numbers between 0 and 255, to address a computer.

IP addresses are normally written as four numbers separated by a period, like this: 192.168.1.50.

Each computer or device must have a unique IP address before it can connect to the Internet.

Each IP packet must have an address before it can be sent to another computer.

In computer terms, TCP/IP uses 32 bits addressing. It uses 4 bytes. One byte is 8 bits. One byte can contain 256 different values:

00000000, 00000001, 00000010, 00000011, 00000100, 00000101, 00000110, 00000111, 00001000 .....and all the way up to 11111111. For example, rtfm.mit.edu is the domain of a particular computer at MIT. The IP of rtfm.mit.edu is 18.181.0.24 (Harley Hahn, 1996)

Subnet Mask:

On most networks, the network administrator uses an IP address scheme that include a custom subnet mask. Because the routes that are in the physical Local Area Network (LAN) define each network, a network administrator must use a different network address for each side of a route. When a router is used to create smaller networks, the smaller network is called subnet. (Andrew G. Blank)

An IP address has two components, the network address and the host address. A subnet mask separates the IP address into the network and host addresses. Subletting further divides the host part of an IP address into a subnet and host address if additional subnetwork is needed. Use the Subnet Calculator to retrieve subnetwork information from IP address and Subnet Mask. It is called a subnet mask because it is used to identify network address of an IP address by performing a bitwise AND operation on the netmask.

A Subnet mask is a 32-bit number that masks an IP address, and divides the IP address into network address and host address. Subnet Mask is made by setting network bits to all "1"s and setting host bits to all "0"s. Within a given network, two host addresses are reserved for special purpose, and cannot be assigned to hosts. The "0" address is assigned

a network address and "255" is assigned to a broadcast address, and they cannot be assigned to hosts.

2. CHAPTER TWO: REVIEW OF LITERATURE

2.1 Modbus Protocol

The Modbus protocol was developed in 1979 by Modicon, Incorporated, for industrial automation systems and Modicon programmable controllers. It has since become an industry standard method for the transfer of discrete/analog I/O information and register data between industrial control and monitoring devices. Modbus is now a widely-accepted, open, public-domain protocol that requires a license, but does not require royalty payment to its owner.

Modbus devices communicate using a master-slave (client-server) technique in which only one device (the master-client) can initiate transactions (called queries). The other devices (slaves or servers) respond by supplying the requested data to the master, or by taking the action requested in the query. A slave is any peripheral device (I/O transducer, valve, network drive, or other measuring device) which processes information and sends its output to the master using Modbus. The Acromag I/O Modules are the slave/server devices, while a typical master device is a host computer running appropriate application software. Other devices may function as both clients (masters) and servers (slaves) (Figure 2.1).

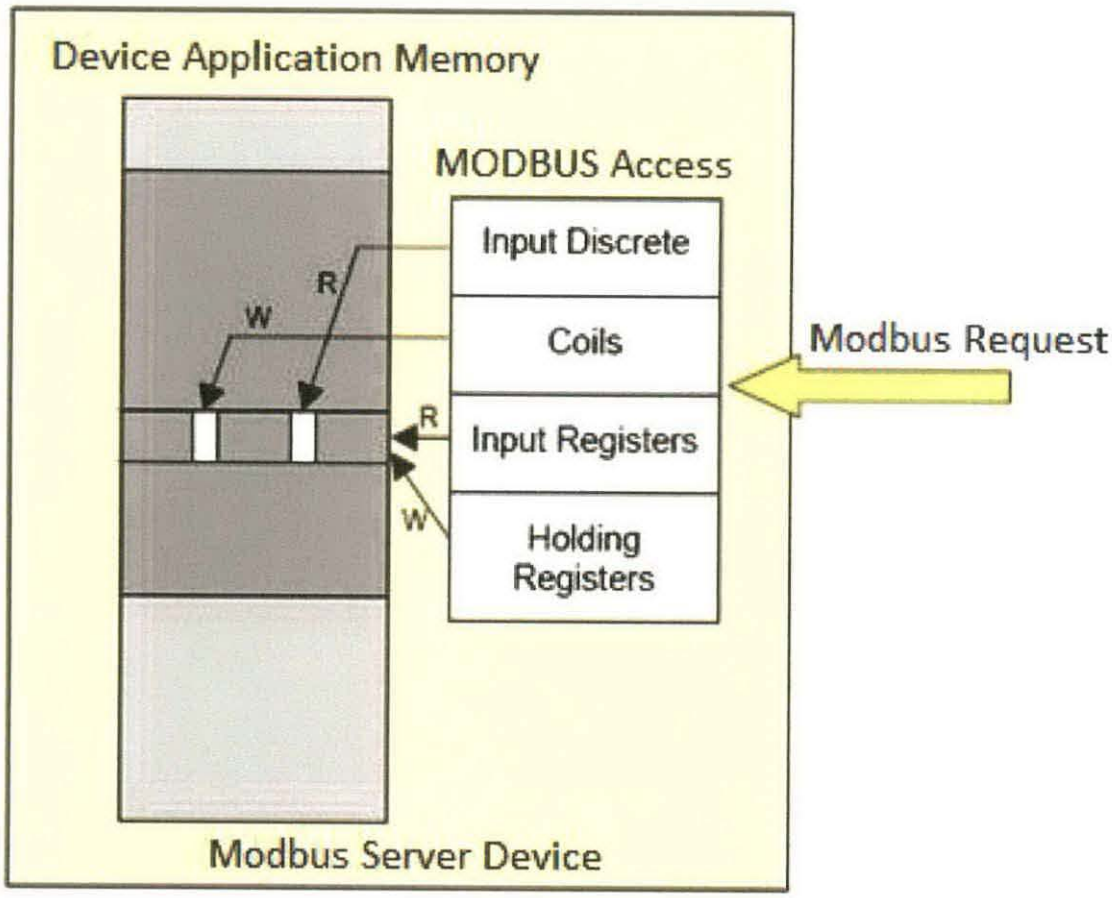


Figure 2.1 Modbus Protocol

The data type of Modbus protocol (Figure 2.2):

Primary tables	Object type	Type of
Discretes Input	Single bit	Read-Only
Coils	Single bit	Read-Write
Input Registers	16-bit word	Read-Only
Holding Registers	16-bit word	Read-Write

Figure 2.2 Modbus Protocol Data Type

Masters can address individual slaves, or can initiate a broadcast message to all slaves.

Slaves return a response to all queries addressed to them individually. Slaves do not initiate messages on their own, they only respond to queries from the master (Figure 2.3).

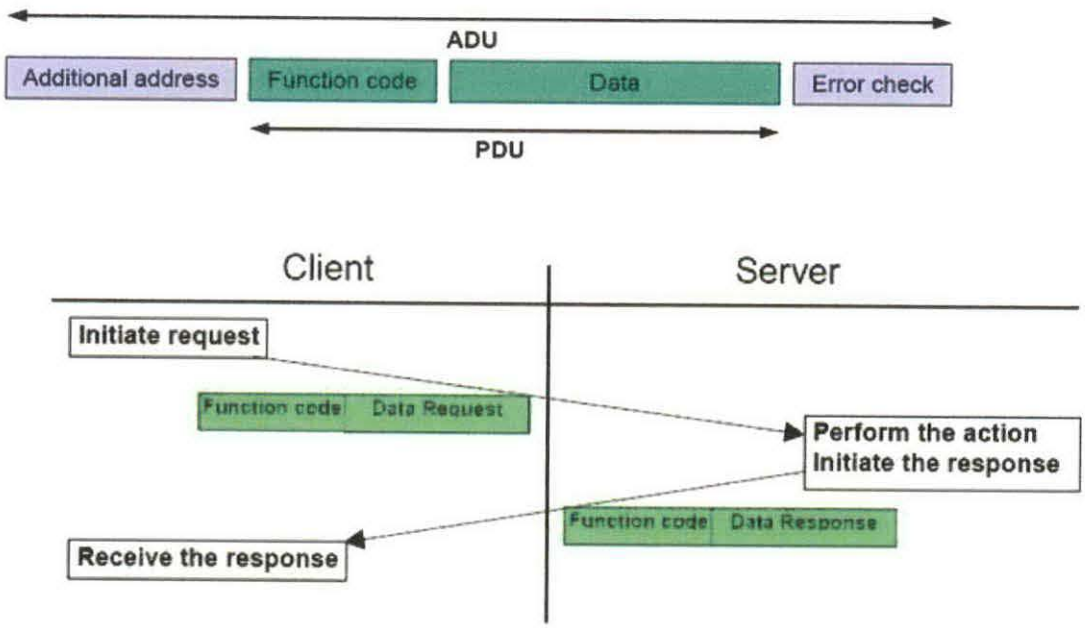


Figure 2.3 Modbus Protocol communication

A master's query will consist of a slave address (or broadcast address), a function code defining the requested action (Figure 2.4), any required data, and an error checking field. A slave's response consists of fields confirming the action taken, any data to be returned, and an error checking field. Note that the query and response both include a device address, a function code, plus applicable data, and an error checking field. If no error occurs, the slave's response contains the data as requested. If an error occurs in the query received, or if the slave is unable to perform the action requested, the slave will return an exception message as its response.

The error check field of the slave's message frame allows the master to confirm that the contents of the message are valid. Traditional Modbus messages are transmitted serially

and parity checking is also applied to each transmitted character in its data frame.

(Acromag, Inc.2005)

Command	Function Code
01	Read Coils
02	Read Discrete Inputs
03	Read Holding Registers
04	Read Input Registers
05	Write Single Coil
06	Write Single Register
07	Read Exception Status
08	Diagnostics
.	
.	
xx	Up to 255 function codes, depending on the device

Figure 2.4 Modbus Protocol Function code

2.2 Modbus/TCP Protocol

The Modbus messaging structure is the application protocol that defines the rules for organizing and interpreting the data independent of the data transmission medium.

TCP/IP refers to the Transmission Control Protocol and Internet Protocol, which provides the transmission medium for Modbus TCP/IP messaging. Simply stated, TCP/IP allows blocks of binary data to be exchanged between computers. It is also a world-wide

standard that serves as the foundation for the World Wide Web. The primary function of TCP is to ensure that all packets of data are received correctly, while IP makes sure that messages are correctly addressed and routed. Note that the TCP/IP combination is merely a transport protocol, and does not define what the data means or how the data is to be interpreted (this is the job of the application protocol, Modbus in this case).

So in summary, Modbus TCP/IP uses TCP/IP and Ethernet to carry the data of the Modbus message structure between compatible devices. That is, Modbus TCP/IP combines a physical network (Ethernet), with a networking standard (TCP/IP), and a standard method of representing data (Modbus as the application protocol). Essentially, the Modbus TCP/IP message is simply a Modbus communication encapsulated in an Ethernet TCP/IP wrapper (Figure 2.6).

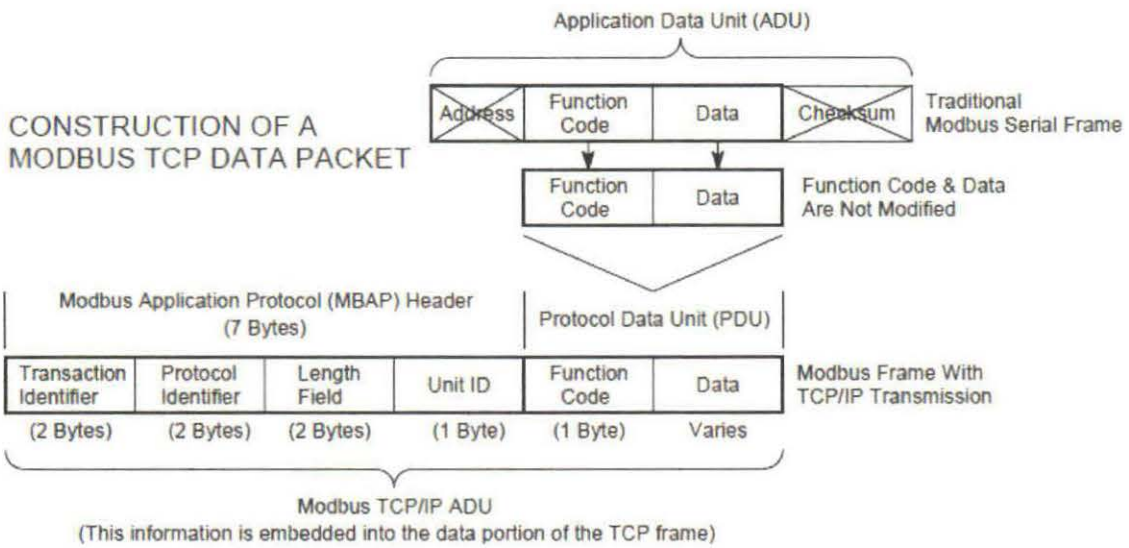


Figure 2.5 Modbus/TCP Protocol

In practice, Modbus TCP embeds a standard Modbus data frame into a TCP frame (Figure 2.6), without the Modbus checksum, as shown in the following diagram.

Fields	Length	Description -	Client	Server
Transaction Identifier	2 Bytes	Identification of a MODBUS Request / Response transaction.	Initialized by the client	Recopied by the server from the received request
Protocol Identifier	2 Bytes	0 = MODBUS protocol	Initialized by the client	Recopied by the server from the received request
Length	2 Bytes	Number of following bytes	Initialized by the client (request)	Initialized by the server (Response)
Unit Identifier	1 Byte	Identification of a remote slave connected on a serial line or on other buses.	Initialized by the client	Recopied by the server from the received request

Figure 2.6 Modbus/TCP Frame

For example, imagine the case writing a coil at address 0x1 to false. The Modbus/TCP Frame will be constructed as follows:

TransI D	Protocol ID	Length	Unit ID	FunCode	Address	Data
0001	0000	0006	00	05	0001	0000

2.3 Big Endian

Big Endian Byte Order: The most significant byte (the "big end") of the data is placed at the byte with the lowest address (Figure 2.7). The rest of the data is placed in order in the next three bytes in memory.

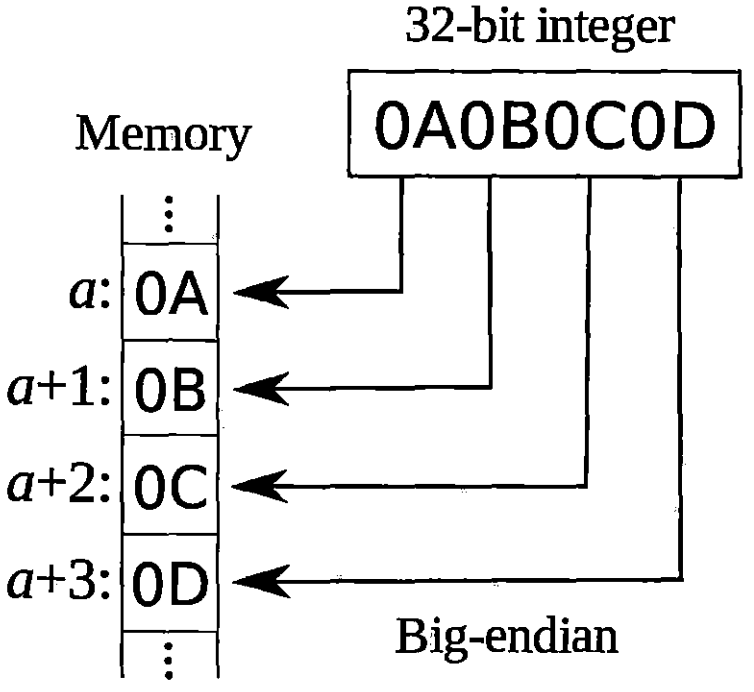


Figure 2.7 Big-endian

Little Endian Byte Order: The least significant byte (the "little end") of the data is placed at the byte with the lowest address (Figure 2.8). The rest of the data is placed in order in the next three bytes in memory.

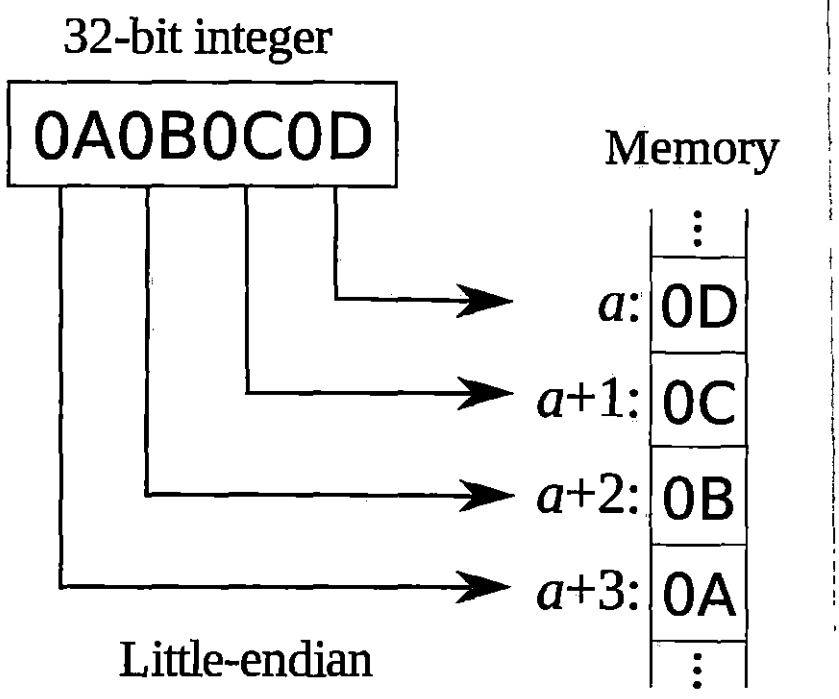


Figure 2.8 Little-endian

Big endian is more natural to most people and thus makes it easier to read hex dumps. By having the high-order byte come first, the developer can always test whether the number is positive or negative by looking at the byte at offset zero. (Linda Null, 2014)

Modbus uses Big-Endian to transfer data. For Windows and Intel system, Little-Endian is used for data presentation and data transfer. So the order of bytes needed to be adjusted before sending and after receiving by Modbus protocol.

The C# code to convert Big-Endian to Litter Endian format is presented:

```
Byte[] data  = new byte[2];

Byte[] byteData = new byte[2];

// get little-endian bytes
```

```
byteData = BitConverter.GetBytes(dataWillBeTransfer);

data[0] = byteData[1];

data[1] = byteData[0];
```

2.4 Client-Server communication mode

The client–server model (Figure 2.9) of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers, which await incoming requests. (Client–server model)

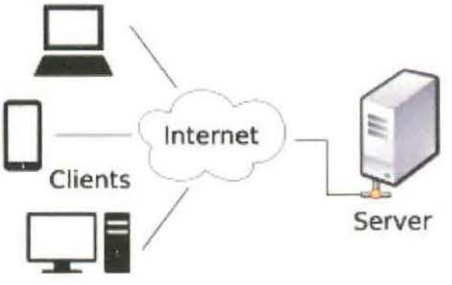


Figure 2.8 the client–server model

2.5 EasyModbusTCP Library

This is the Modbus-TCP library for .NET and Java implementations. The library is suitable for Client and Server applications. The simulator shows registers and coils. They can be changed within the server simulator. Also protocol information can be displayed, to

monitor the data exchange between Server and Client.

The following function Codes are supported:

- Read Coils (FC1)
- Read Discrete Inputs (FC2)
- Read Holding Registers (FC3)
- Read Input Registers (FC4)
- Write Single Coil (FC5)
- Write Single Register (FC6)
- Write Multiple Coils (FC15)
- Write Multiple Registers (FC16)
- Read/Write Multiple Registers (FC23)

(EasyModbusTCP Library http://www.easymodbustcp.net/)

Figure 2.10 The Server Simulator

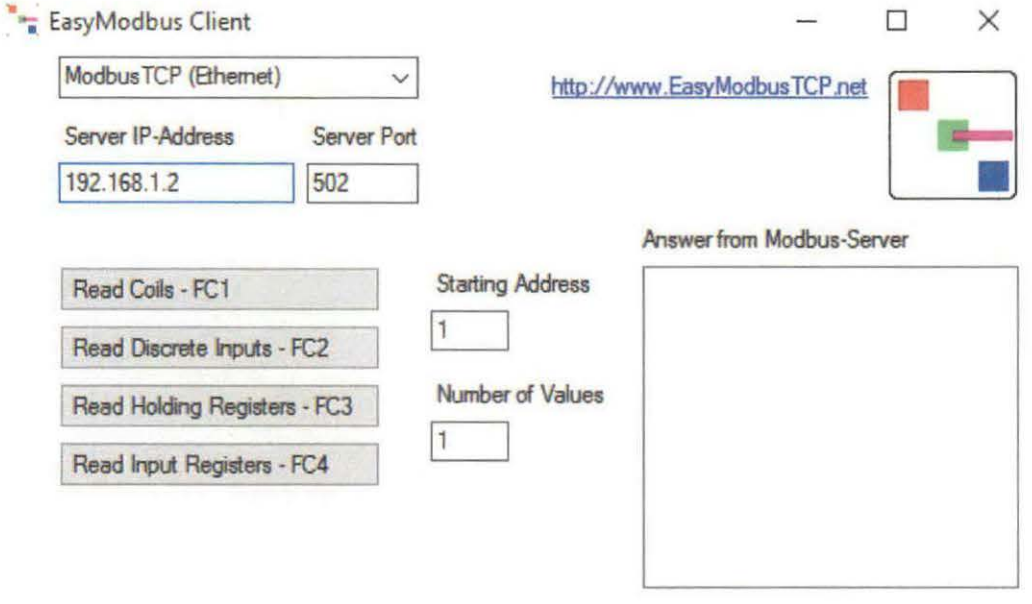


Figure 2.11 Client Simulator

By reading the source code of Modbus-TCP library and using function codes provided by the server simulator and client simulator, we can understand the implementation of Modbus/TCP at the server side and the client side.

2.6 Modbus/TCP implementation in C#

The key point of the Modbus communication is to build the data frame (Figure 2.12) for the protocol according to the specification of the protocol:

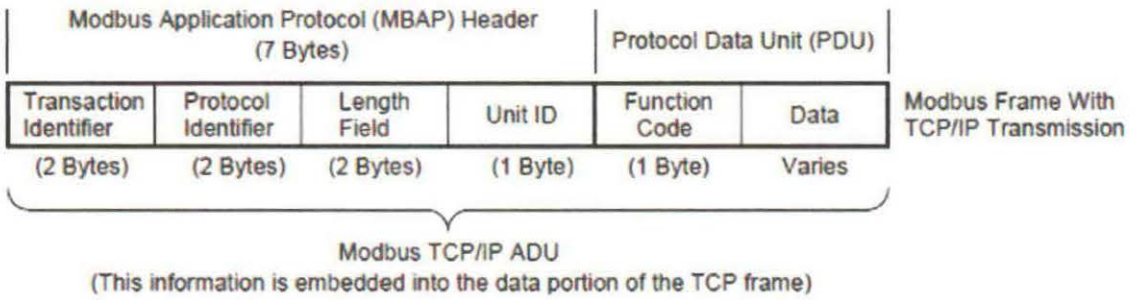


Figure 2.12 Modbus Frame

The processes of building the Modbus/TCP frame (Figure 2.13)

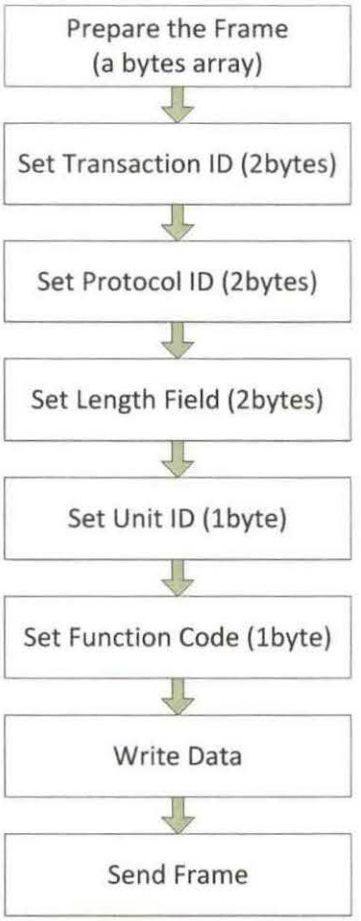


Figure 2.13 Processes of building the Modbus/TCP frame

2.6.1 Implementation of Modbus/TCP at Client Side

Here is some pseudo-code for writing a single coil to a server device (http://easymodbustcp.net/)

```
/// <summary>
/// Write single Coil to Master device (FC5).
/// </summary>
/// <param name="startingAddress">Coil to be written</param>
/// <param name="value">Coil Value to be written</param>
```

```
public void WriteSingleCoil(int startingAddress, bool value)
{
   byte[] coilValue = new byte[2];
   this.transactionIdentifier = BitConverter.GetBytes((int)0x0001);
   this.protocolIdentifier = BitConverter.GetBytes((int)0x0000);
   this.length = BitConverter.GetBytes((int)0x0006);
   this.functionCode = 0x05;
   this.startingAddress = BitConverter.GetBytes(startingAddress);
   if (value == true)
   {
      coilValue = BitConverter.GetBytes((int)0xFF00);
   }
   else
   {
      coilValue = BitConverter.GetBytes((int)0x0000);
   }
   Byte[] data = new byte[]{    this.transactionIdentifier[1],
                                this.transactionIdentifier[0],
                                this.protocolIdentifier[1],
                                this.protocolIdentifier[0],
                                this.length[1],
                                this.length[0],
                                this.unitIdentifier,
```

```
                                        this.functionCode,
                                        this.startingAddress[1],
                                        this.startingAddress[0],
                                        coilValue[1],
                                        coilValue[0]
                };
        stream.Write(data, 0, data.Length);
        data = new Byte[2100];
        stream.Read(data, 0, data.Length);

        if (data[7] == 0x85 & data[8] == 0x01)
            throw new Exception("Function code not supported by master");
        if (data[7] == 0x85 & data[8] == 0x02)
            throw new Exception("Starting address invalid or starting address + quantity invalid");
        if (data[7] == 0x85 & data[8] == 0x03)
            throw new Exception("quantity invalid");
        if (data[7] == 0x85 & data[8] == 0x04)
            throw new Exception("error reading");
}
```

Read a single coil:

```
/// <summary>
/// Read Coils from Master device (FC1).
/// </summary>
```

```
/// <param name="startingAddress">First coil to be read</param>
/// <param name="quantity">Numer of coils to be read</param>
/// <returns>Boolean Array which contains the coils</returns>
public bool[] ReadCoils(int startingAddress, int quantity)
{
   bool[] response;
   this.transactionIdentifier = BitConverter.GetBytes((int) 0x0001);
   this.protocolIdentifier = BitConverter.GetBytes((int) 0x0000);
   this.length = BitConverter.GetBytes((int)0x0006);
   this.functionCode = 0x01;
   this.startingAddress = BitConverter.GetBytes(startingAddress);
   this.quantity = BitConverter.GetBytes(quantity);
   Byte[] data = new byte[]{
                this.transactionIdentifier[1],
                this.transactionIdentifier[0],
                this.protocolIdentifier[1],
                this.protocolIdentifier[0],
                this.length[1],
                this.length[0],
                this.unitIdentifier,
                this.functionCode,
                this.startingAddress[1],
                this.startingAddress[0],
```

```
            this.quantity[1],
            this.quantity[0],
};
stream.Write(data, 0, data.Length);
data = new Byte[2100];
stream.Read(data, 0, data.Length);

if (data[7] == 0x81 & data[8] == 0x01)
   throw new Exception("Function code not supported by master");
if (data[7] == 0x81 & data[8] == 0x02)
   throw new Exception("Starting address invalid or starting address + quantity invalid");
if (data[7] == 0x81 & data[8] == 0x03)
   throw new Exception("quantity invalid");
if (data[7] == 0x81 & data[8] == 0x04)
   throw new Exception("error reading");

response = new bool[quantity];
for (int i = 0; i < quantity; i++)
{
   int intData = data[9+i/8];
   int mask = Convert.ToInt32(Math.Pow(2, (i%8)));
   response[i] = Convert.ToBoolean((intData & mask)/mask);
}
```

```
        return (response);
}
```

2.6.2 Implementation of the Modbus/TCP at the Server Side

Here is some pseudo-code for writing a single coil to a server device (http://easymodbustcp.net/)

```
    private void ReadCoils(ModbusProtocol receiveData, ModbusProtocol sendData, NetworkStream
stream, int portIn, IPAddress ipAddressIn)
        {
            sendData.response = true;
            sendData.transactionIdentifier = receiveData.transactionIdentifier;
            sendData.protocolIdentifier = receiveData.protocolIdentifier;
            sendData.unitIdentifier = receiveData.unitIdentifier;
            sendData.functionCode = receiveData.functionCode;
            if ((receiveData.quantity < 1) | (receiveData.quantity > 0x07D0))  //Invalid quantity
            {
                sendData.errorCode = (byte)(receiveData.functionCode + 0x80);
                sendData.exceptionCode = 3;
            }
            if ((receiveData.startingAdress + 1 + receiveData.quantity) > 65535)   //Invalid Starting adress
or Starting address + quantity
            {
                sendData.errorCode = (byte)(receiveData.functionCode + 0x80);
                sendData.exceptionCode = 2;
```

```
            }
            if ((receiveData.quantity % 8) == 0)
              sendData.byteCount = (byte)(receiveData.quantity / 8);
            else
              sendData.byteCount = (byte)(receiveData.quantity / 8 + 1);

            sendData.sendCoilValues = new bool[receiveData.quantity];
            Array.Copy(coils, receiveData.startingAdress + 1, sendData.sendCoilValues, 0,
receiveData.quantity);

            if (true)
            {
              Byte[] data;
              if (sendData.exceptionCode > 0)
                data = new byte[9];
              else
                data = new byte[9 + sendData.byteCount];
              Byte[] byteData = new byte[2];

              sendData.length = (byte)(data.Length - 6);

              //Send Transaction identifier
              byteData = BitConverter.GetBytes((int)sendData.transactionIdentifier);
              data[0] = byteData[1];
```

```
data[1] = byteData[0];


//Send Protocol identifier

byteData = BitConverter.GetBytes((int)sendData.protocolIdentifier);

data[2] = byteData[1];

data[3] = byteData[0];


//Send length

byteData = BitConverter.GetBytes((int)sendData.length);

data[4] = byteData[1];

data[5] = byteData[0];


//Unit Identifier

data[6] = sendData.unitIdentifier;


//Function Code

data[7] = sendData.functionCode;


//ByteCount

data[8] = sendData.byteCount;


if (sendData.exceptionCode > 0)

{
```

```
    data[7] = sendData.errorCode;
    data[8] = sendData.exceptionCode;
    sendData.sendCoilValues = null;
}

if (sendData.sendCoilValues != null)
    for (int i = 0; i < (sendData.byteCount); i++)
    {
        byteData = new byte[2];
        for (int j = 0; j < 8; j++)
        {

            byte boolValue;
            if (sendData.sendCoilValues[i * 8 + j] == true)
                boolValue = 1;
            else
                boolValue = 0;
            byteData[1] = (byte)((byteData[1]) | (boolValue << j));
            if ((i * 8 + j + 1) >= sendData.sendCoilValues.Length)
                break;
        }
        data[9 + i] = byteData[1];
    }
```

```
            stream.Write(data, 0, data.Length);
        }
```

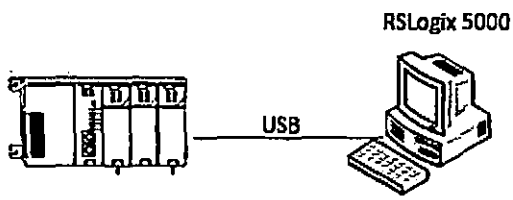
2.7 Modbus TCP Sample Application

Rockwell Automation provides a sample application to demonstrate the use of the Modbus TCP Master Sample Application. The program was written to run on a Control Logix 5000 processor with a compatible Ethernet Module. (Modbus TCP Master Sample Application Revision 1.02)

3. CHAPTER THREE: METHODOLOGY AND FINDINGS

3.1 Setup development environment

The RSLogix 5000 is the PLC programming environment be used for PLC programming. The RSLogix 5000 is running on the computer and using RSLinx Classic to upload program to the PLC and control the PLC remotely.



(Figure 3.1) PCL development environment

3.2 Build physical connection

Firstly, the PLC should be connected with the RJ45 connector of the Ethernet cable to one of the Ethernet ports on the controller. The ports are on the bottom of the controller.

(Figure 3.2) (CompactLogix 5370 Controllers User Manual)

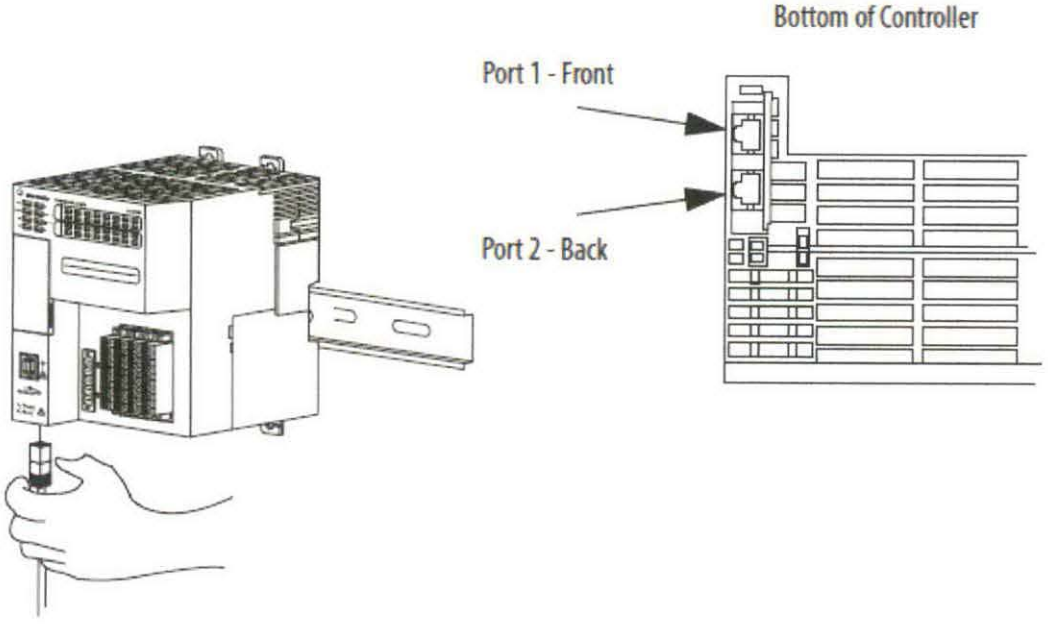


Figure 3.2 CompactLogix 5370 Ethernet port

The PLC can communicate with the robot or virtual robot through Modbus/TCP. (Figure 3.2)

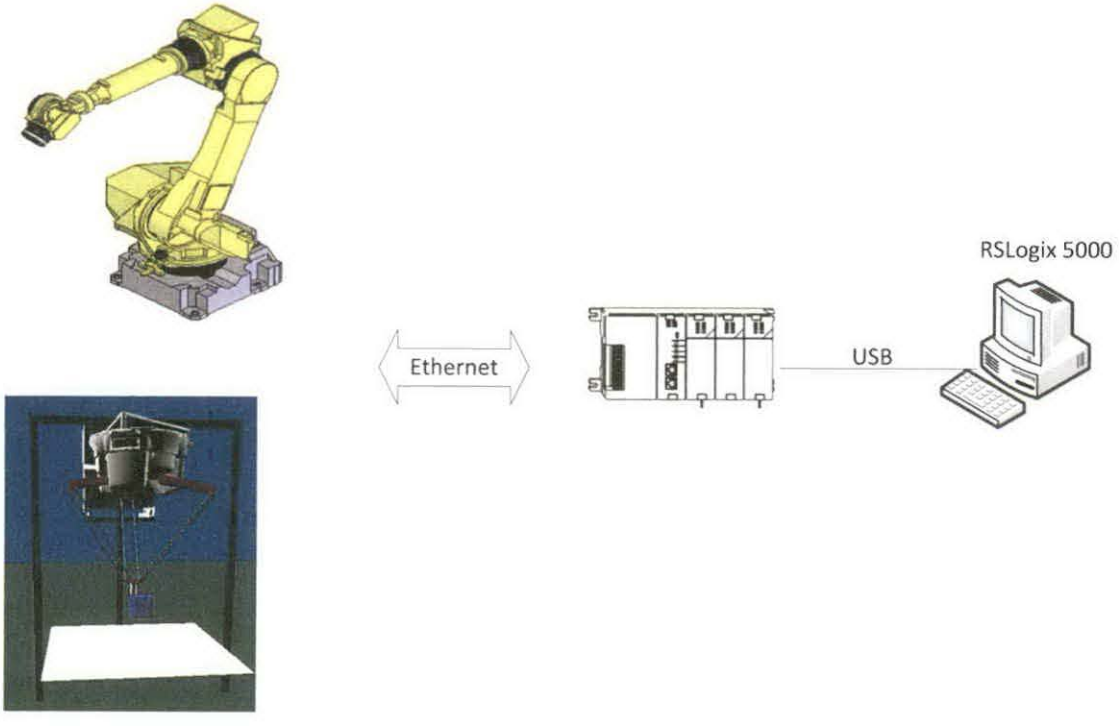


Figure 3.2 Communication between the PLC and the Robot

Any device connected to an Ethernet should have an IP address.

If the virtual emulation and PLC connect to an Ethernet by a router (Figure 3. 3), they can use the DHCP (Dynamic Host Configuration Protocol) Server to get the IP address dynamically. It is very convenient to connect the device to Ethernet. However, the disadvantage is that the IP address would be changed when the devices restart. That means the PLC program needs to change the IP address before building a connection with the virtual robot.

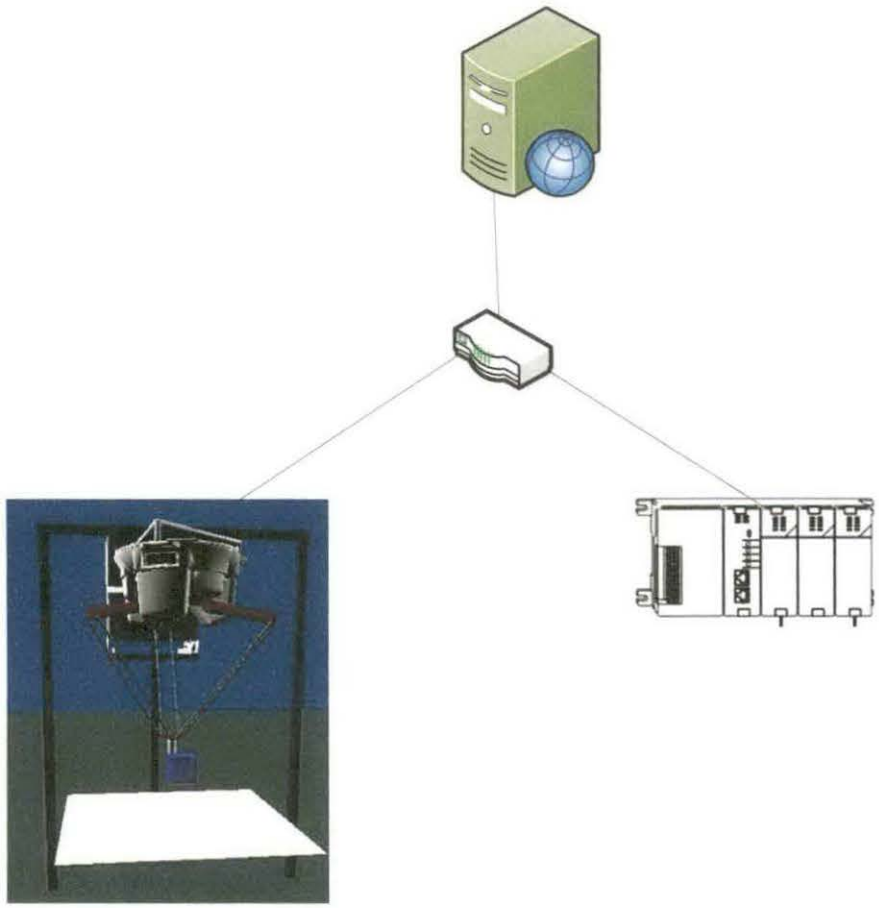


Figure 3.3 Connect PLC with Virtual Robot Through Router

The other solution is to connect the virtual robot and the PLC with a crossover cable (Wikipedia https://en.wikipedia.org/wiki/Ethernet_crossover_cable) and setup IP addresses for them manually.

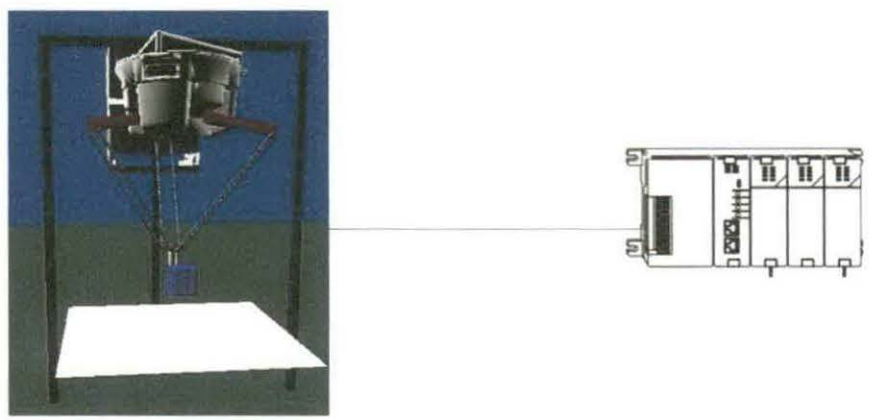


Figure 3.4 Connecting the PLC to the Virtual Robot Through Crossover Cable

In RSLogix 5000, one set up the IP address for the PLC. (Figure 3.5)

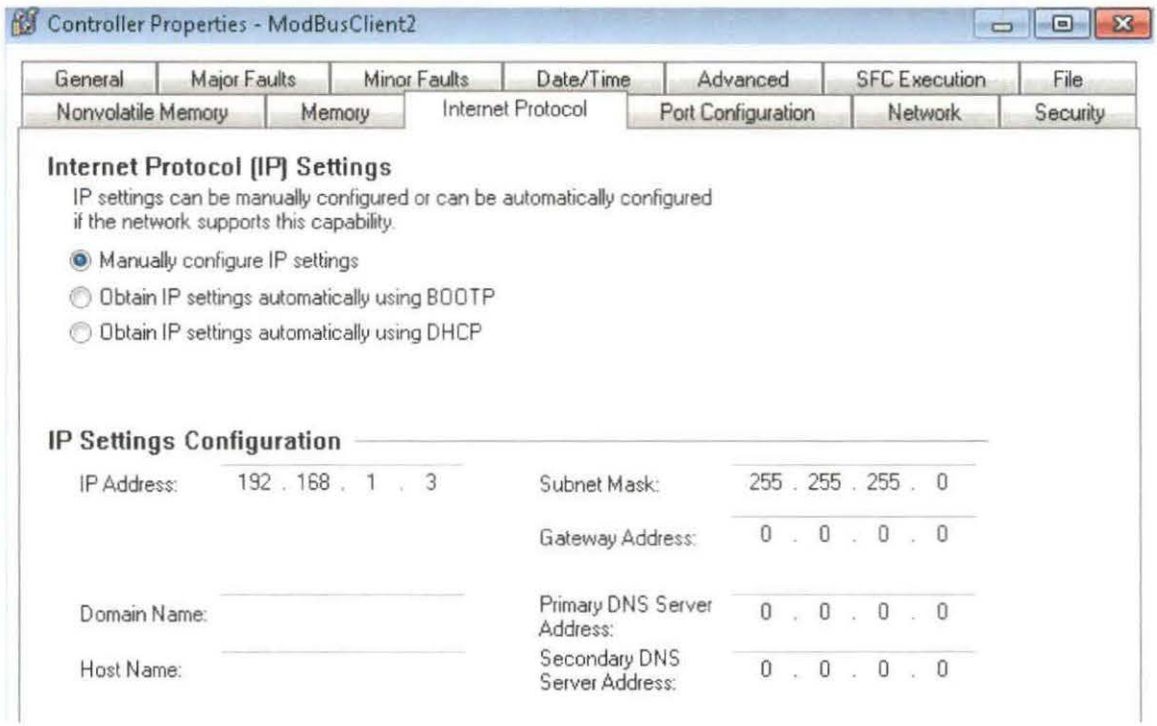


Figure 3.5 Set IP address for PLC

3.3 Research Utilities

In this research, to investigate the data in Modbus/TCP frame, a software utility was created to covert the float number to integer number (Figure 3.6). By comparing the result created by this software utility and the value in the PLC or virtual robot, we can know whether the data transfer is correct or not.

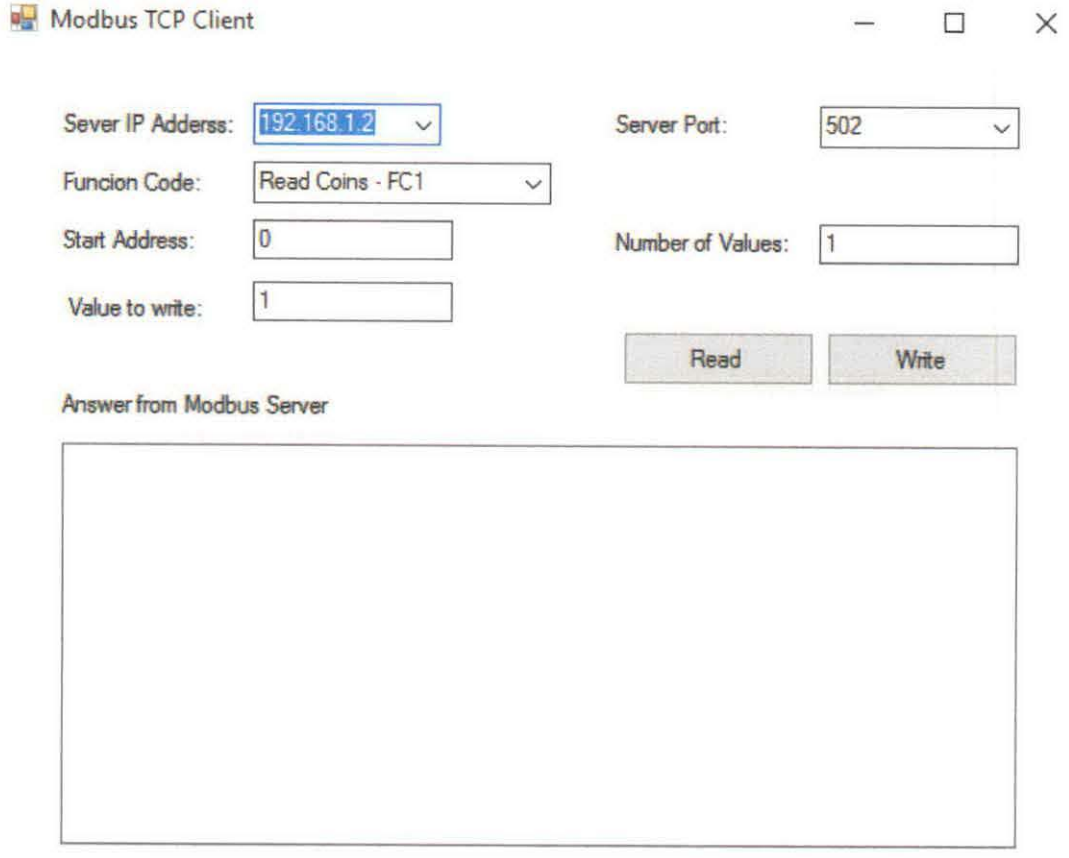


Figure 3.7 Modbus TCP Client

3.4 PLC Programming

The stored instruction set that is programmed into a PLC is called the work cycle program. The work cycle program is derived for the program of instructions and /or the process flow for the application being controlled. A work cycle program is created by dividing and processing the sequential list of action specified by the program of instructions into logic and sequencing instructions. (Daniel E. Kandray, P.E, 2010.)

The PLC ladder diagram is used for programming. On the ladder diagram, Input instructions or output instruction were put on each rung.

The Modbus Client program is based on the sample application provided by Rockwell Automation, Inc. (Modbus TCP Master Sample Application Revision 1.02)

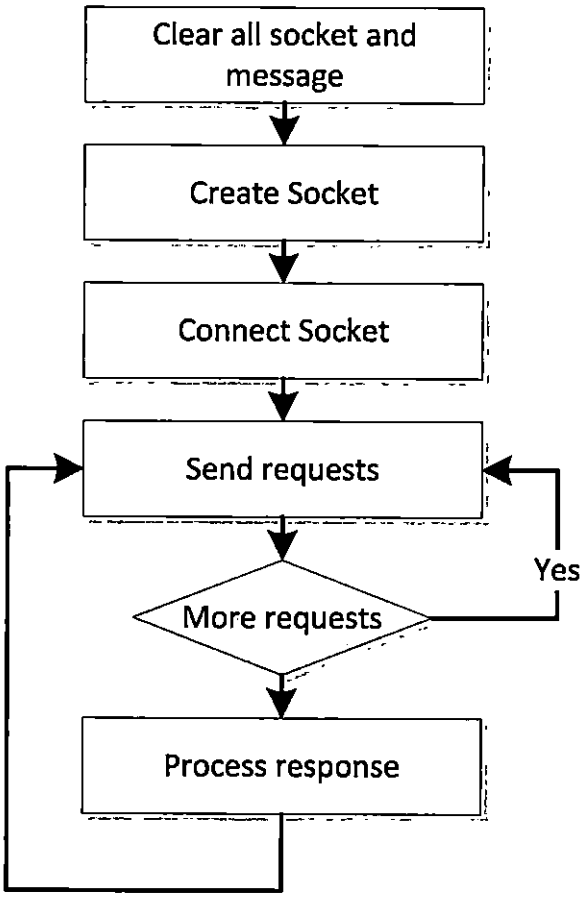


Figure 3.8 PLC Program Process

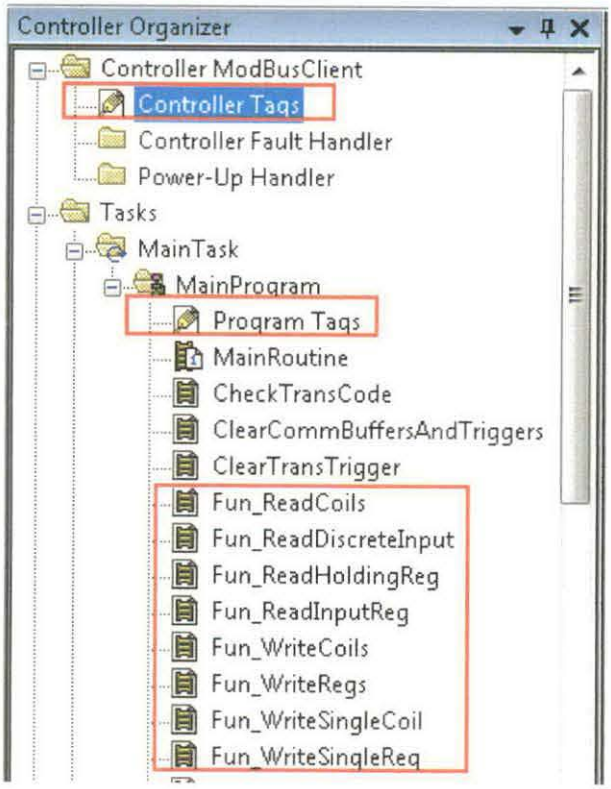


Figure 3.9 The PLC Program

The MSG instruction (Figure 3.10) asynchronously reads or writes a block of data to another module on a network. The PLC program uses it to create a connection, send data and retrieve data. After the connection is built, the PLC program can read and write data from the server through the Modbus protocol.

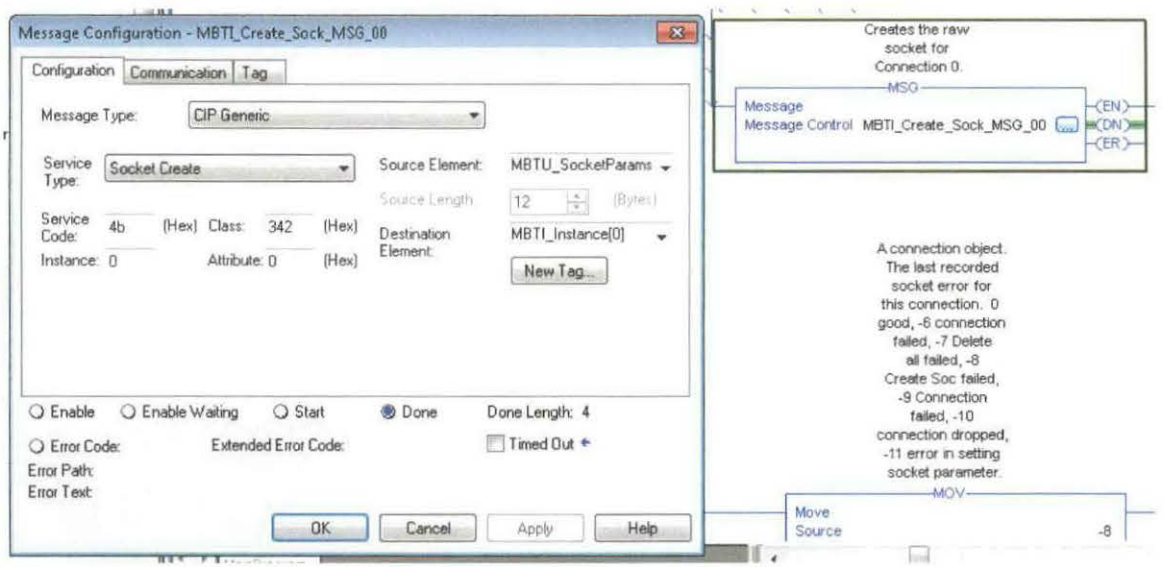


Figure 3.10 The MSG Instruction

Before the data is sent out, the PLC program needs to prepare data by filling some data structure.

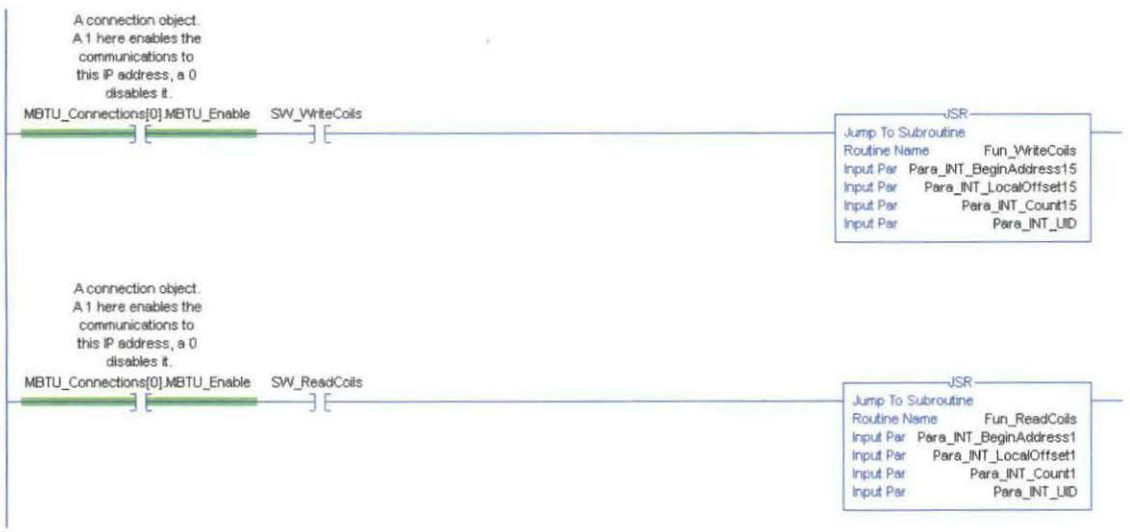


Figure 3.11 The PLC Program for data preparing

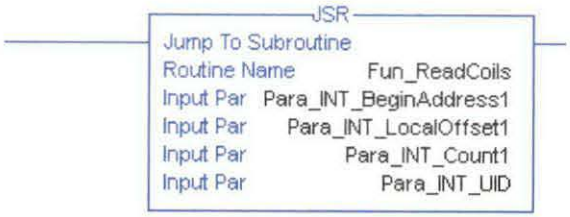


Figure 3.12 PLC Route

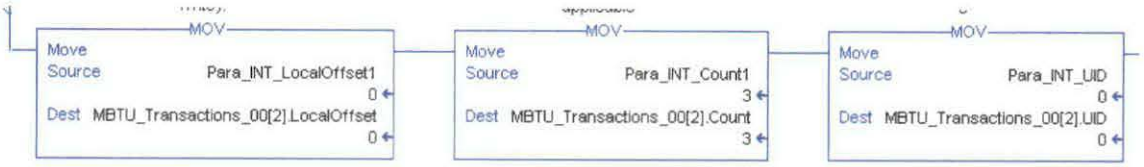


Figure 3.13 Set Route Parameter

In the PLC program, data structure MBTranscation is used to send requests.

MBTU_Transactions_00	{...}
MBTU_Transactions_00[0]	{...}
MBTU_Transactions_00[0].BeginAddress	11
MBTU_Transactions_00[0].Count	1
MBTU_Transactions_00[0].Enabled	0
MBTU_Transactions_00[0].LocalOffset	11
MBTU_Transactions_00[0].PollInterval	2
MBTU_Transactions_00[0].ReqBuilt	0
MBTU_Transactions_00[0].Request	'-$C1$00$...
MBTU_Transactions_00[0].TransComplete	1
MBTU_Transactions_00[0].TransID	11713
MBTU_Transactions_00[0].TransLastError	-2
MBTU_Transactions_00[0].TransStat	0
MBTU_Transactions_00[0].TransType	5
MBTU_Transactions_00[0].UID	100

Figure 3.14 MB Transaction

Every MB transaction (Figure 3.15) corresponds to a request. The transaction data structure works for a Modbus/TCP frame, which contains all the information used to fill the frame.

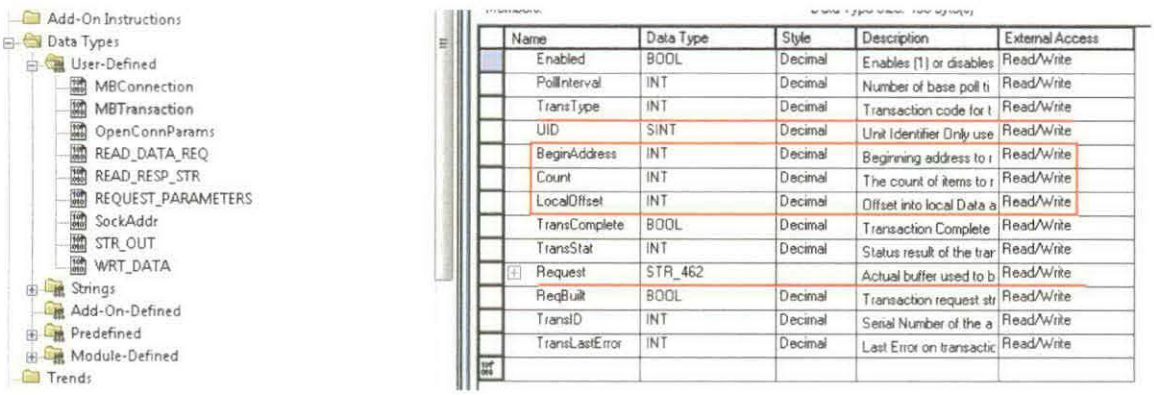


Figure 3.15 Data in MB Transaction

The PLC program fills MB Transaction according to the configuration, if the Enabled field of MB Transaction is true, MB Transaction will be sent by MSG instruction. That is one communication between PLC and the server.

In the PLC program, some data buffers were used for data reading and writing

Data buffer	Type	Usage
MBTU_MB_0xx	BOOL[1024]	For function code 1(Read coils), 5(Write single coil) and 15(Write multiple coils)
MBTU_MB_1xx	BOOL[1024]	For function code 2(Read discrete coils)
MBTU_MB_3xx	INT[256]	For function code 4(Read input coils)
MBTU_MB_4xx	INT[256]	For function code 3(Read holding register), 6(Write single register) and 16(Write multiple registers)

For example, in the following instruction

Field Name	Value
TransType	1
BeginAddress	0
Count	10
LocalOffset	1

TransType means a coil reading.

BeginAddress is 0; this means the request will start reading at the first coil the device.

The count is 10; this means 10 coil will be read.

LocalOffset is 1; this means the data will be put into the buffer starting at the address 1.

3.4.1 Multi Tasks

A Logix5000 controller support the following types of tasks:

Continuous Task: The continuous task runs all the time in the background. When it completes a full scan, it restarts immediately. A project does not require a continuous task. If used a project can have only one continuous task. Therefore, the main task of the project is a continuous task.

Periodic Task: A periodic task performs at a specific period. The time period can be set from 0.1ms to 2000s. The default period is 10ms.

Event Task: An event task performs only when a specific event occurs.

(Allen-Bradley, Logix5000 Controllers Tasks, Programs and Routines)

The PLC program has two tasks in the project (Figure 3.16), one is for communication and one for the data process.

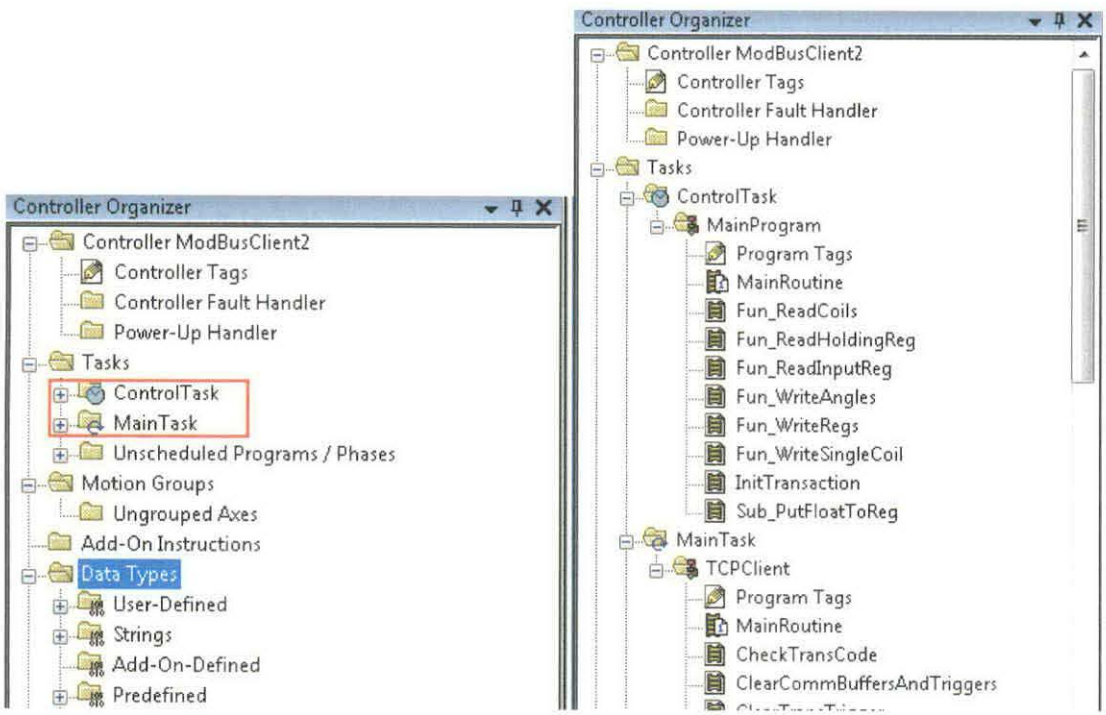


Figure 3.16 PLC Tasks

When a PLC wants to send a request to the virtual robot, the data process task prepares data and puts the data into global data structures. The global data structures are some data structures be used to shard data between control task and communication task. The communication task scans the global data structure continuously. If it finds the data is ready, the communication task will send data to the virtual robot.

When a PLC wants to read data from the virtual robot, the data process task prepares the request and puts it into the global data structure. If the communication finds there is a request in the global data structure, the communication task will send a request to the virtual robot and write the response from the virtual robot into the global data structure. Then the data process task can read data locally.

According to this design, the communication task has higher priority than the data process task. So the communication task was set as the main task and the data process task was set as periodic task (Figure 3.17).

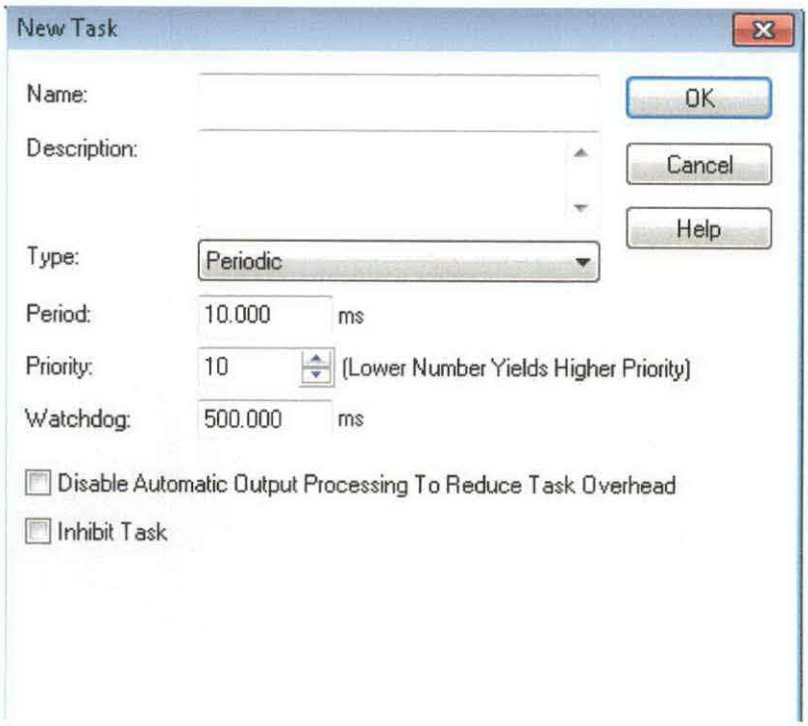


Figure 3.17 Tasks type

3.4.2 Task synchronization

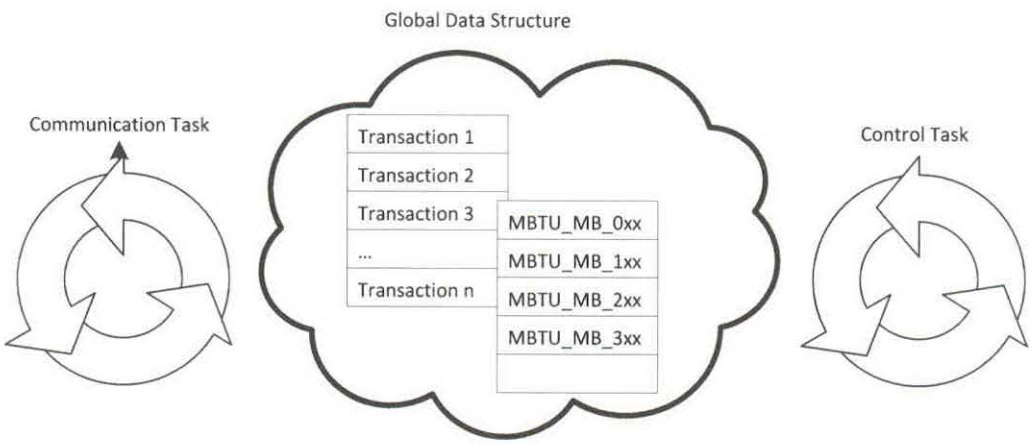


Figure 3.18 Tasks synchronization

As the main task, the communication task scans the global data structure and

communicates with the virtual robot continuously. At the same time, the data process task performs at a specific period. If the period of data process task is shorter than the period of communication task, some requests in the global data structure will be overwritten by the subsequent request. It is an example of a producer–consumer problem in computer science.

To solve this problem, there are two sample solutions:

1. Use some flag or trigger. Let the communication task notify the data process task, after it has finished, read the global data structure and send request.

2. Make the period of data process task long enough. In this period, the data process task will be suspended and the communication task has enough time to read the global data structure and send the request.

Currently, the second solution was used in this research. Because it is easier to be implemented.

3.5 Data Processing

There are two types of data used for the virtual robot operation, Boolean and Float. Processing Boolean data is straight-forward and extra convert is need for Float data.

The virtual robot keeps the value of the position as the Float number, which is 32 bit. But the input register for Modbus/TCP can keep just an integer value. The Float data in virtual will be treated as two integer numbers. Furthermore, the Modbus uses big endian data format so the order of the bytes of the data will be respected.

3.5.1 Read Float

For example, there is a Float number in the virtual robot that will be read through

Modbus/TCP protocol. Because the virtual robot is running on a windows system, which uses little endian data format, the Float number is represented in the memory as following: (Figure 3.18)

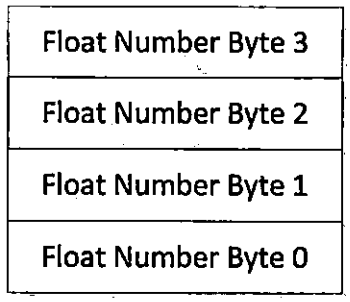


Figure 3.18 Float Number in Memory

Before the number was put into the Modbus/TCP protocol frame, the bytes of the number should be reversed to big endian format:

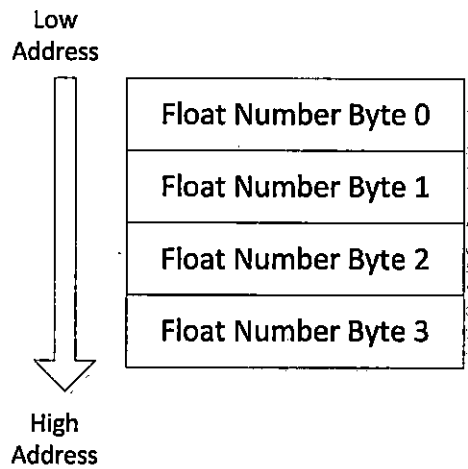


Figure 3.19 Float Bytes in the Memory

The bytes will be put into Modbus/TCP protocol frame and read by the PLC. After the Modbus frame was received by PLC, the PLC program will treat the 4 bytes as two integer

number and reverse them to little endian format.

Because the PLC is running on a Windows system and uses a little endian data format, the PLC will treat the data as a two 16 bits integer number, the bytes will be converted to:

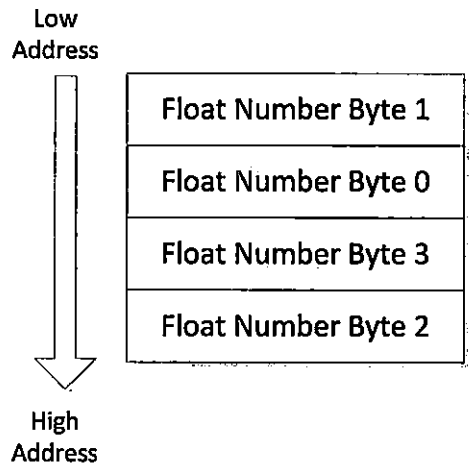


Figure 3.20 Float bytes in memory

To read the data correctly, the PLC program needs to re-arrange the bytes to the original order.

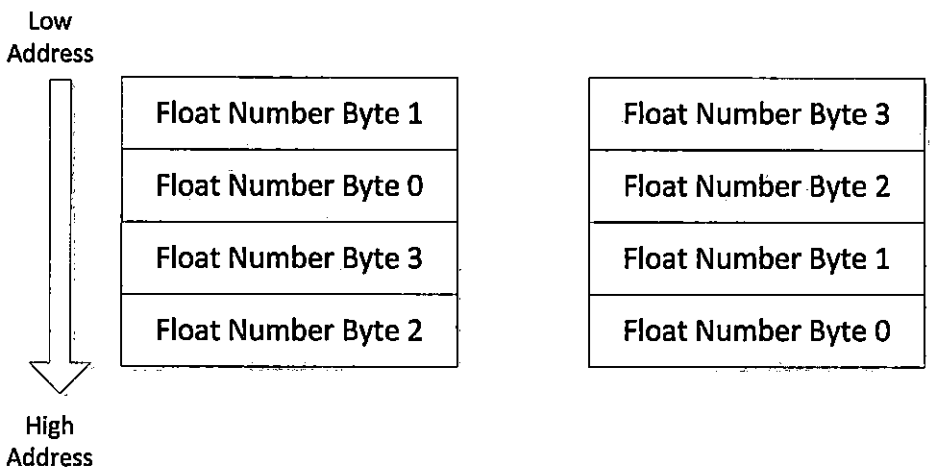


Figure 3.21 Reverse the Bytes

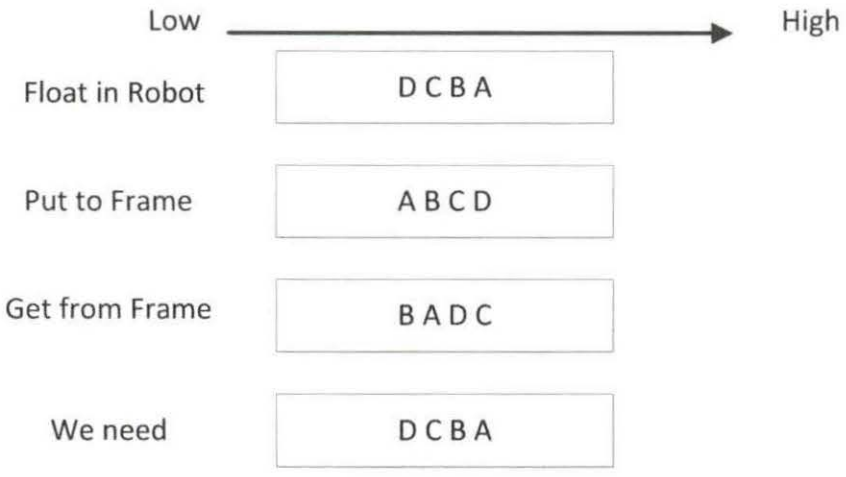


Figure 3.22 reverse the bytes

The code looks like following:

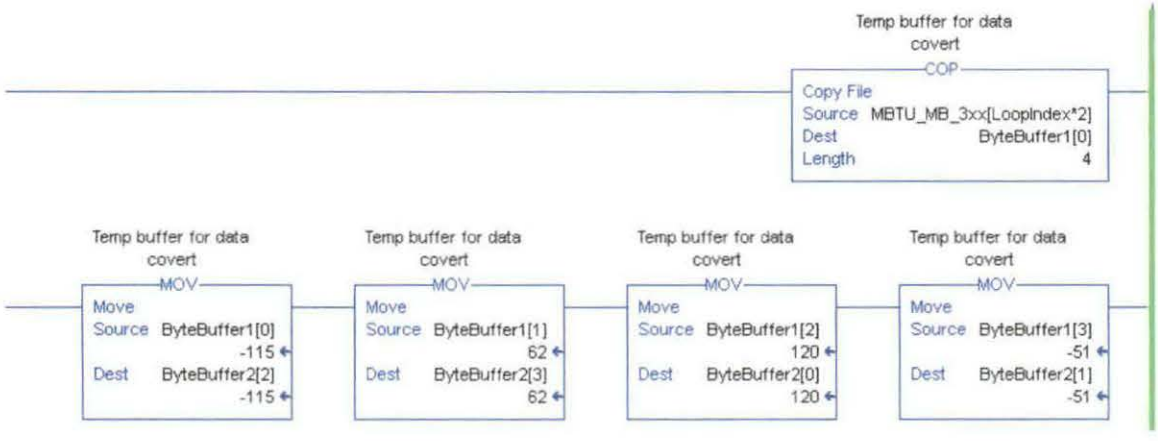


Figure 3.22 PLC Program

3.5.2 Write Float

To write the Float number to the virtual robot, the PLC program will split the four bytes of the number to two parts, each part having two bytes.

Firstly, the four bytes were put into a byte array.

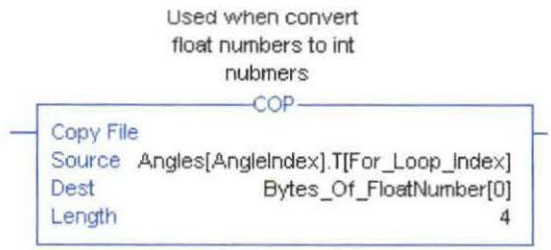


Figure 3.23 Float Number in PLC

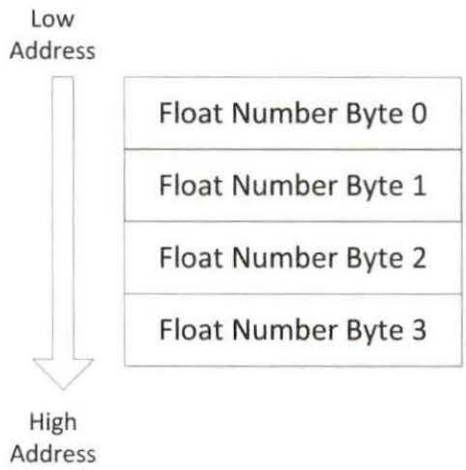


Figure 3.24 Bytes of Float Number

After that, the PLC program reverses those two parts to big endian order (Figure 3.23).

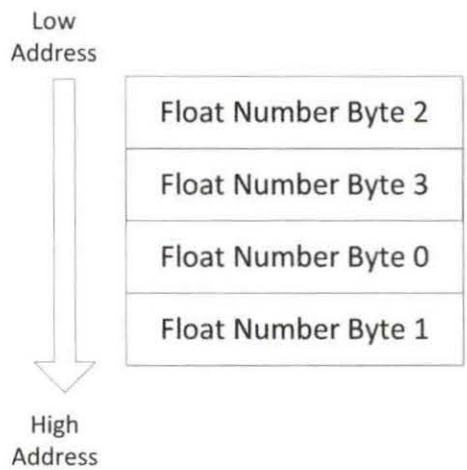


Figure 3.23 Reverse of the Bytes

The PLC ladder diagram is:

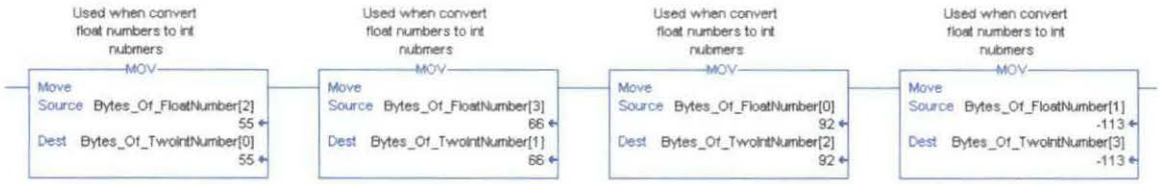


Figure 3.24 PLC Program

Then the data will be sent to the virtual robot.

When the bytes were sent to the virtual robot, the TCP/IP library will think the bytes are two integers with big endian format. The bytes will be re-ordered as follows:

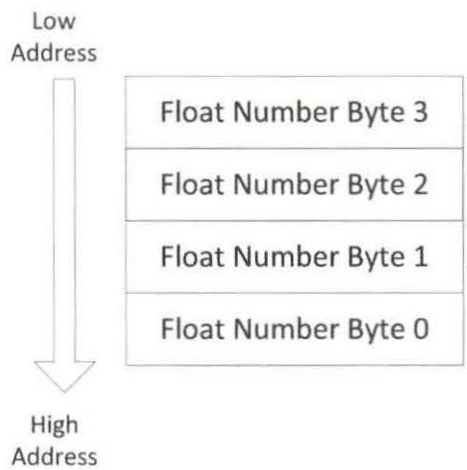


Figure 3.25 Bytes of Float Number

The virtual robot knows it is a Float with four bytes in big endian format. So the virtual robot will reverse the four bytes into little endian and read it correctly.

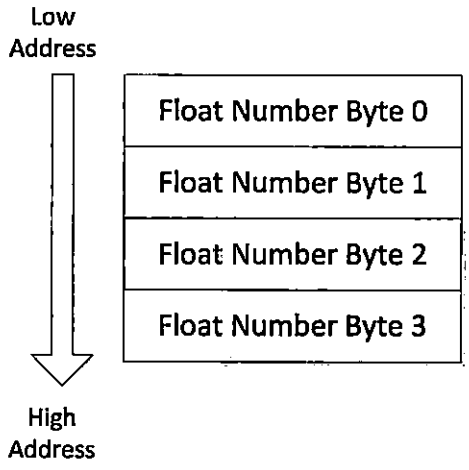


Figure 3.26 bytes of float number

3.6 Functions

The functions are some subroutines used to prepare and receive data.

Every function has 4 parameters:

Name:	Meaning
BeginAddress	The start address of the data to be read or write
LocalOffset	The address of data will be saved locally in data array MBTU_MB_XXX
Count	Counter of the bytes to read or write
UID	The ID of the device to be operate

Global data structure MBTU_MB0XX is a Boolean array, and is used to read and write coils. The corresponding function code are 1(which means read coils), 5(which means write single coil), and 15(which means write multiple coils)

Global data structure MBTU_MB1XX is Boolean array, it is used to read discrete inputs. The corresponding function code is 2 (which means read discrete inputs)

Global data structure MBTU_MB3XX is 16 bits integer array, which is used to read input registers. The corresponding function code is 4 (which means read input registers)

Global data structure MBTU_MB4XX is 16 bits integer array, which is used to read and write holding registers. The corresponding function code are 3(which means read holding registers), 6(which means write single holding register), and 16(which means write multiple registers).

In the reading operation, the communication task will read data from virtual robot and save them to those global data structure so control task can read it.

In the writing operation, the control task will prepare data and put them into those global data structures so control task can send them to the virtual robot.

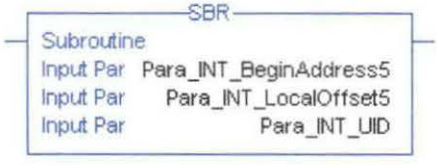
3.6.1 Fun_ReadCoils

The function sets function code to 1 in the transaction data structure.

Transaction_00[2] and read serval coils (Boolean type) from target device. The parameter Begin Address means the function will start reading at address of the device being reading.

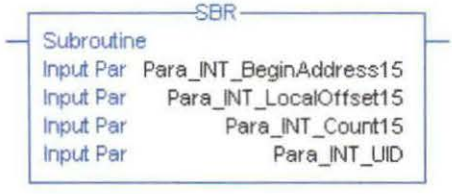
The data read from target device will be put into Boolean array MBTU_MB0XX.

3.6.2 Fun_WriteSingleCoils



This function sets function code to 5 in the transaction data structure. Transaction_00[6] sends the Boolean value in MBTU_MB0XX at address of LocalOffixedata to the virtual robot's coil at address BeginAddress.

3.6.3 Fun_WriteCoils



This function sets function code to 15 in the transaction data structure Transaction_00[1] and sends some Boolean values in MBTU_MB0XX starting from address of LocalOffixedata to the virtual robot's coil at address BeginAddress.

3.6.4 Fun_ReadDiscreteInput

The function sets function code to 2 in the transaction data structure Transaction_00[3] and reads serval discrete input from target device. The parameter Begin Address means the function will start reading at the address of the selected device. The data read from target device will be put into Boolean array MBTU_MB1XX.

3.6.5 Fun_ReadInputReg

The function sets function code to 4 in the transaction data structure Transaction_00[3] and reads serval input register from target device. The parameter Begin Address means the function will start reading at the address of the selected device. The data read from target device will be put into Boolean array MBTU_MB3XX.

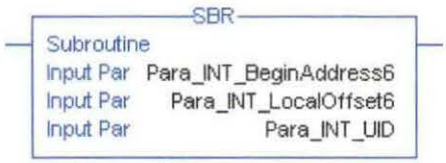
3.6.6 Fun_ReadHoldingReg

The function sets function code to 3 in the transaction data structure Transaction_00[4] and reads serval holding registers (16 bits integer type) from target

device. The parameter Begin Address means the function will start reading at the address of the selected device.

The data read from target device will be put into Boolean array MBTU_MB4XX.

3.6.7 Fun_WriteSingleReg



This function sets function code to 6 in the transaction data structure Transaction_00[6] and sends the 16 bits integer value in MBTU_MB4XX at the address of LocalOffixedata to the virtual robot's holding register at address BeginAddress.

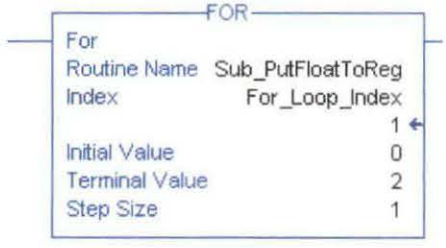
3.6.8 Fun_WriteRegs

This function sets function code to 16 in the transaction data structure Transaction_00 [7] and sends the 16 bits integer value in MBTU_MB4XX at the address of LocalOffixedata to the virtual robot's holding register at address BeginAddress. This function will convert the bytes of the Float to big endian format.

3.6.9 Fun_WriteAngles

Each angle of the virtual robot is composed by three Float numbers which have six

bytes in all. Firstly, this function will convert the float numbers to six bytes with big endian order. The bytes will be put into a data table named "MBTU_MB_4xx"



Then, this function will be called function "Fun_WriteRegs". "Fun_WriteRegs" reads the data in "MBTU_MB_4xx" to send the six bytes located at address 0 to 5 to the virtual robot.

3.6.10 Process_Transaction

This function will scan the content of the transaction data structure from Transaction_00[0] to Transaction_00[7]. Each transaction data structure stands for a function code. If Process_Transaction finds one transaction data structure is ready, Process_Transaction builds a Modbus/TCP frame according to the content of the transaction data structure and sends a request to the target device.

3.7 Action Sequence

To make the virtual robot accomplish an action sequence, Control task communicated with the communication task through the shared global data structure (Figure 3.27).

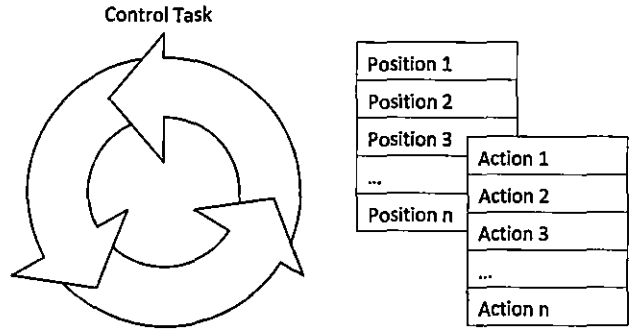


Figure 3.27 Action Sequence

The communication task reads the status of the virtual robot and saves the data in those global data structure.

Control task will scan the position list and action list and prepare the frame data, and put the data in the global data structure. When the data is ready, the communication will send the request to the server.

4. CONCLUSION AND RECOMMENDATIONS

The purpose of this research was to introduce the virtual robot and other virtual devices into teaching. From the beginning, the focus of this research was to implement the communication between computer and the PLC through the Modbus/TCP protocol.

As the result, a PLC and computer network was built and a PLC Modbus client program was created to communicate with the virtual robot running on the PC by sending and retrieving data from the computer. The PLC program can control the virtual robot moving around, opening or closing a grabber, and determine whether the virtual robot has reached the intended target object or not. The action sequence of the robot can be

enhanced by adding more positions and actions to the current PLC program.

Some future work may be necessary in this research to support more virtual devices in the emulation system.

In the future, by using the Modbus/TCP protocol, the PLC can control multiple virtual robots running in the emulation system or real robots in the industrial environment (Figure 3.28).

The trainer can teach students how to control the devices by using a PLC and allow them to practice with the virtual device without worrying about damage to the real devices and/or themselves.

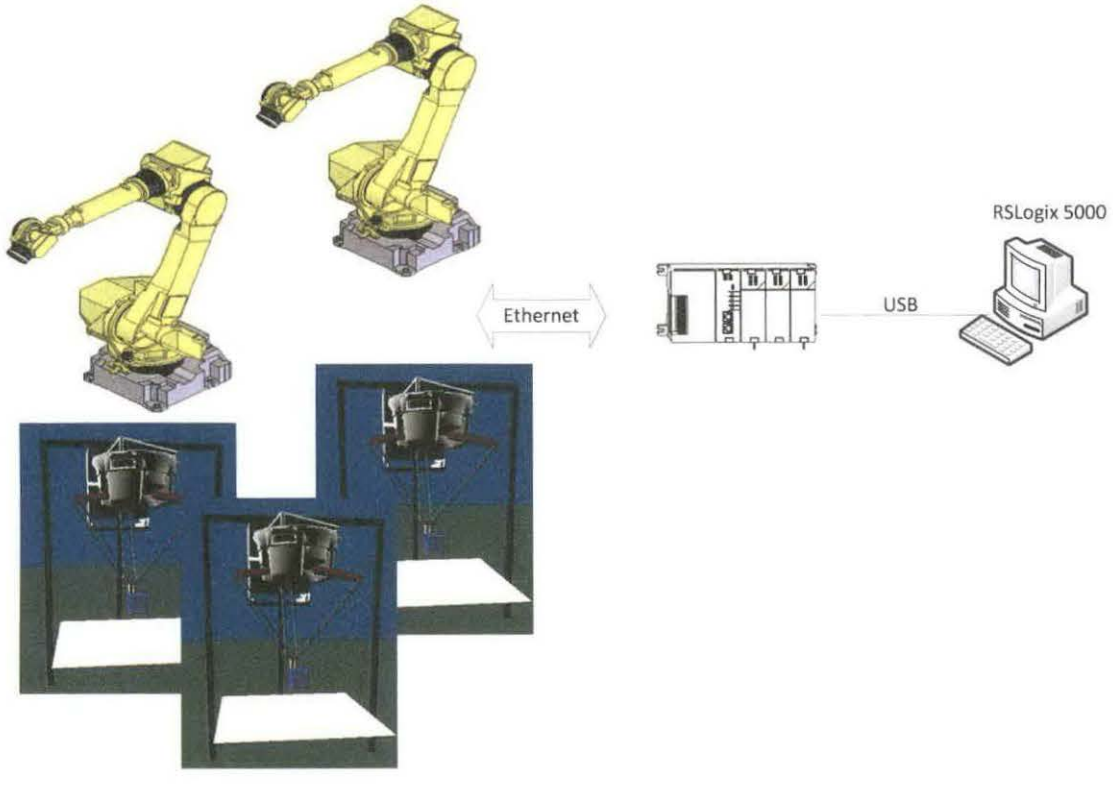


Figure 3.28 Control Multiple Devices

5. REFERENCES

Daniel E. Kandray, P.E, 2010. Programmable Automation Technologes- An Introduction to CNC, Robotics and PLCs. (p. 343)

Zyda, M., 2005, "From visual simulation to virtual reality to games", Computer, (p. 25-32)

John W. Webb, 1995. Programmable logic Controllers Principles and Applications (3rd ed., p. 4). Merrill publishing Company.

Kenneth G. Oliver, 1990. Basic Industrial Electricity: A Training and Maintenance Manual (p. 44). Industrial press.

Sunit Kumar Sen, 2014. Fieldbus and Networking in Process Automation (p.185). CRC Press

June Jamrich Parsons, 2014. New Perspectives on Computer Concepts (p. 257)

Acromag, Inc.2005, Introduction to Modbus TCP/IP (p.4)

John Wiley & Sons, 2013. Networking for Dummies (10 ed., 6-3)

Harley Hahn, 1996. The Internet Complete reference. (p.70)

Andrew G. Blank, 2004. TCP/IP Foundations. (p.128)

What is a Subnet Mask? (https://www.iplocation.net/subnet-mask)

Daniel E. Kandray, P.E, 2010. Programmable Automation Technologes- An Introduction to CNC, Robotics and PLCs. (p. 385)

Acromag, Inc.2005, Introduction to Modbus TCP/IP (p.4)

Charles Petaold, 1998. Programming Windows (5ed, ch23.) Microsoft.

Jeffery Richter, 2012. CLR via C# (4ed, p.23)

Linda Null, 2014. The Essentials of Computer Organization and Architecture(p. 297)

EasyModbusTCP Library (http://www.easymodbustcp.net/)

An Introduction to Socket Programming in .NET using C# (http://www.codeproject.com/Articles/10649/An-Introduction-to-Socket-Programming-in-NET-using)

Allen-Bradley, Logix5000 Controllers Tasks, Programs and Routines (p9)

6. APPENDIX Source Code related

Here is some pseudo-code about writing a single coil to a master device (http://easymodbustcp.net/)

```
/// <summary>
/// Write single Coil to Master device (FC5).
/// </summary>
/// <param name="startingAddress">Coil to be written</param>
/// <param name="value">Coil Value to be written</param>
public void WriteSingleCoil(int startingAddress, bool value)
{
   byte[] coilValue = new byte[2];
   this.transactionIdentifier = BitConverter.GetBytes((int)0x0001);
   this.protocolIdentifier = BitConverter.GetBytes((int)0x0000);
   this.length = BitConverter.GetBytes((int)0x0006);
   this.functionCode = 0x05;
```

```
this.startingAddress = BitConverter.GetBytes(startingAddress);
if (value == true)
{
   coilValue = BitConverter.GetBytes((int)0xFF00);
}
else
{
   coilValue = BitConverter.GetBytes((int)0x0000);
}
Byte[] data = new byte[]{    this.transactionIdentifier[1],
                             this.transactionIdentifier[0],
                             this.protocolIdentifier[1],
                             this.protocolIdentifier[0],
                             this.length[1],
                             this.length[0],
                             this.unitIdentifier,
                             this.functionCode,
                             this.startingAddress[1],
                             this.startingAddress[0],
                             coilValue[1],
                             coilValue[0]
          };
stream.Write(data, 0, data.Length);
```

```
        data = new Byte[2100];
        stream.Read(data, 0, data.Length);

        if (data[7] == 0x85 & data[8] == 0x01)
            throw new Exception("Function code not supported by master");
        if (data[7] == 0x85 & data[8] == 0x02)
            throw new Exception("Starting address invalid or starting address + quantity invalid");
        if (data[7] == 0x85 & data[8] == 0x03)
            throw new Exception("quantity invalid");
        if (data[7] == 0x85 & data[8] == 0x04)
            throw new Exception("error reading");
}
```

Read a single-coil:

```
    /// <summary>
    /// Read Coils from Master device (FC1).
    /// </summary>
    /// <param name="startingAddress">First coil to be read</param>
    /// <param name="quantity">Numer of coils to be read</param>
    /// <returns>Boolean Array which contains the coils</returns>
    public bool[] ReadCoils(int startingAddress, int quantity)
    {
        bool[] response;
        this.transactionIdentifier = BitConverter.GetBytes((int) 0x0001);
```

```
this.protocolIdentifier = BitConverter.GetBytes((int) 0x0000);

this.length = BitConverter.GetBytes((int)0x0006);

this.functionCode = 0x01;

this.startingAddress = BitConverter.GetBytes(startingAddress);

this.quantity = BitConverter.GetBytes(quantity);

Byte[] data = new byte[]{

        this.transactionIdentifier[1],

        this.transactionIdentifier[0],

        this.protocolIdentifier[1],

        this.protocolIdentifier[0],

        this.length[1],

        this.length[0],

        this.unitIdentifier,

        this.functionCode,

        this.startingAddress[1],

        this.startingAddress[0],

        this.quantity[1],

        this.quantity[0],

};

stream.Write(data, 0, data.Length);

data = new Byte[2100];

stream.Read(data, 0, data.Length);
```

```
        if (data[7] == 0x81 & data[8] == 0x01)
            throw new Exception("Function code not supported by master");
        if (data[7] == 0x81 & data[8] == 0x02)
            throw new Exception("Starting address invalid or starting address + quantity invalid");
        if (data[7] == 0x81 & data[8] == 0x03)
            throw new Exception("quantity invalid");
        if (data[7] == 0x81 & data[8] == 0x04)
            throw new Exception("error reading");

        response = new bool[quantity];
        for (int i = 0; i < quantity; i++)
        {
            int intData = data[9+i/8];
            int mask = Convert.ToInt32(Math.Pow(2, (i%8)));
            response[i] = Convert.ToBoolean((intData & mask)/mask);
        }
        return (response);
}
```