Demonstration of Modbus Protocol for Robot Communication Using C#

Sudip Chakraborty[1] & P. S. Aithal[2]
[1]Post-Doctoral Researcher, College of Computer science and Information science, Srinivas University, Mangalore-575 001, India
OrcidID: 0000-0002-1088-663X; E-mail: sudip.pdf@srinivasuniversity.edu.in
[2]ViceChancellor, Srinivas University, Mangalore, India
OrcidID: 0000-0002-4691-8736; E-Mail: psaithal@gmail.com

Subject Area: Robotics.
Type of the Paper: Experiment-based Research.
Type of Review: Peer Reviewed as per |C|O|P|E| guidance.
Indexed In: OpenAIRE.
DOI: https://doi.org/10.5281/zenodo.5709235
Google Scholar Citation: IJAEML

How to Cite this Paper:
Sudip Chakraborty, & Aithal, P. S., (2021). Demonstration of Modbus Protocol for Robot Communication Using C#. *International Journal of Applied Engineering and Management Letters (IJAEML)*, *5*(2), 119-131.
DOI: https://doi.org/10.5281/zenodo.5709235

International Journal of Applied Engineering and Management Letters (IJAEML)
A Refereed International Journal of Srinivas University, India.

Crossref DOI : https://doi.org/10.47992/IJAEML.2581.7000.0108

© With Authors.

This work is licensed under a Creative Commons Attribution-Non-Commercial 4.0 International License subject to proper citation to the publication source of the work.
Disclaimer: The scholarly papers as reviewed and published by the Srinivas Publications (S.P.), India are the views and opinions of their respective authors and are not the views or opinions of the S.P. The S.P. disclaims of any harm or loss caused due to the published content to any party.

Demonstration of Modbus Protocol for Robot Communication Using C#

Sudip Chakraborty[1] & P. S. Aithal[2]
[1]Post-Doctoral Researcher, College of Computer science and Information science, Srinivas University, Mangalore-575 001, India
OrcidID: 0000-0002-1088-663X; E-mail: sudip.pdf@srinivasuniversity.edu.in
[2]ViceChancellor, Srinivas University, Mangalore, India
OrcidID: 0000-0002-4691-8736; E-Mail: psaithal@gmail.com

ABSTRACT

Purpose: *The Modbus is the trusted name in the industrial automation communication domain. It is a pretty simple protocol to implement and so very popular to the industrial communication personnel. Nowadays, some industrial robots are also capable of communicating through Modbus. So, our robot researchers frequently face the challenge of communicating with Modbus-enabled devices or robots. They need to know the protocol in detail before integrating it into their project. Its learning curves are a bit higher because of the lack of document which is practical oriented. The protocol selection, packet structure, CRC, or LRC calculation need to maintain precisely as standards; otherwise, the Modbus exception may happen. We experience those scenarios. Through our practical experience, we learned what is required for a new researcher who wants to implement Modbus in their project. In this paper, we demonstrate the Modbus packet structure and implement it with several practical examples. Finally, to test the written code, we provide simple tools which are easy to use and customizable. The researcher can easily integrate into their research project. The complete project source code is available in Github.*

Design/Methodology/Approach: *The Modbus is the standard protocol to communicate between or among the devices. We need a better understanding of it and interface software to test around all aspects. Here we described some practical examples. The GUI is created using C# language inside the Microsoft Visual Studio. The application has several capabilities. In the TCP/IP mode, It can be a server or client. In RTU mode, it can play as a Master or slave device. We can also run two instances in a single system. To communicate between two running apps in RTU mode, we need virtual loopback software, two physical comm port, or two USB to Serial modules. For Modbus TCP mode, we can test within the system using the localhost address (127.0.0.1) or need an IP address for a different.*

Findings/results: *The robot researcher can find helpful information about communicating the robot through the Modbus protocol. The practical example can help them to create packet purser. The functional CRC algorithm code can be used for better understanding and implementation into their project.*

Originality/Value: *This work has some different features than other available utilities. We added features based on our research needs. Our created application is a little bit different from a professional approach. Various display formats are available in our GUI. That makes a difference in the originality of this work. Our GUI can be master, slave, server, or client, which is rarely available.*

Paper Type: *Experiment-based Research.*

Keywords: Modbus Communication, Robot Communication, Modbus RTU master, Modbus RTU client, Modbus TCP Master, Modbus TCP client

1. INTRODUCTION :

Modbus is one of the popular protocols to exchange data between devices in the industrial automation field. In our one project, it was required to implement Modbus protocol to fetch the data from the robot. We started searching the documents on Modbus. Our experience was not very good. There are

enormous documents available on Modbus. But few of them focus on the implementation level. Their paper lacks a correct Packet structure, and in most cases, their CRC algorithm is not valid. Our realization is as below:

- ❖ Most of the documents are incomplete.
- ❖ Less information for practical implementation
- ❖ The software utility is not open-sourced, or code is not available.
- ❖ Most of the software needs to purchase; in most cases, we cannot afford it.
- ❖ In some software is not research-centric, whether they provide little functionality.
- ❖ We didn't find any software to get the data in different formats according to our requirements.

We started our research from here. We achieved our goal. We implement Modbus into our project. After completing our research work, we realized that it should be documented and provided to our robot researcher struggling to implement Modbus into their project. We demonstrated the Modbus protocol and packet structure as practical as possible. We created a GUI utility software for experiment with the Modbus.

2. RELATED WORKS :

Tamboli, S. et al, in their paper, demonstrate how to communicate with an Industrial batch processing system in which the temperature parameter is precisely controlled by several logic. They use Modbus RTU and Modbus TCP for their communication backbone. [1]. Jaloudi, S, in his research, study on polling-based and event-based protocols under Industrial Internet of Things (IIoT) environment. The two scenarios are proposed to build the IIoT environment. The first scenario considers the Modbus TCP as an IoT protocol and builds the environment using the Modbus TCP [2]. G. Ebenhofer et al. present their development approach using development tools and explicitly highlight useful extensions added for a smooth integration of heterogeneous components of various frameworks [3]. H. Lin et al. provide a solution enabling the innovative robotic arm (Staubli) to convert its original communication protocol (SOAP/WSDL) into the role of an OPC UA server. The solution is successfully integrated into ERP/MES manufacturing production systems. The proposed method can help basic IoT robots seamlessly embed in ERP/MES manufacturing systems [4]. Tarek Al-Geddawy presents a method to build a simulated changeable learning factory and link it to the physical system to create a digital twin. The studied learning factory (LEAF) is an open-source low-cost changeable automated system. The method uses the open-source Modbus TCP and OPC UA industrial communication protocols to establish the connection between the physical modules and the digital objects. [5]. M. Hemmatpour et al., in their work, describe the development of a distributed industrial IoT gateway, called DIIG, able to relay industrial network data to a centralized data store. DIIG exploits a real-time client-server programming model based on S7 communication and Modbus TCP protocols [6]. Qingdao Huang et al., in their paper, control the furnace temperature using fuzzy logic. For remote communication, they used the Modbus/TCP industrial Ethernet via OPC [7]. Maldonado V et al., in their research, present the development of a low-cost industrial data server for communicating Modbus TCP devices and Windows applications by using a database as a temporary storage space [8]. X. Xiang et al. present a real-time and reliable communication system for a newly developed hybrid underwater robotic vehicle (HURV), enabling the vehicle's state monitoring and teleoperation mode based on the industrial Ethernet Modbus/TCP via the fiber-optic communication channel [9]. G. Hewei et al. developed a friendlier human-machine interface. They communicate with the manipulator through TCP/IP communication protocol, and the communication is developed using Socket technology under the C++MFC environment [10]. W. Jingran et al., in their paper, focus on the security problems existing in industrial communication protocols, study the security protocols to guarantee the secure transmission of industrial data, and realize the end-to-end identity authentication and data encryption transmission on Modbus/TCP protocol [11]. The above-included papers are suitable to implement the Modbus protocol. But lack some explanation on Modbus in detail which would be helpful for new researchers.

3. OBJECTIVES :

Our object is to provide practical information on Modbus communication to the new researcher for Modbus-enabled robotics communication. Need some learning curve to implement Modbus communication. Here we provide information with practical examples so that anyone can implement Modbus with little effort. For easily understanding, we capture real-world data from Modbus running

device. To test and debug on Modbus, we need one software utility tool. We created this utility tool using C# inside the Microsoft visual studio community edition. Our researchers can test, debug, and finally can be integrated into their research works.

4. APPROACH AND METHODOLOGY :

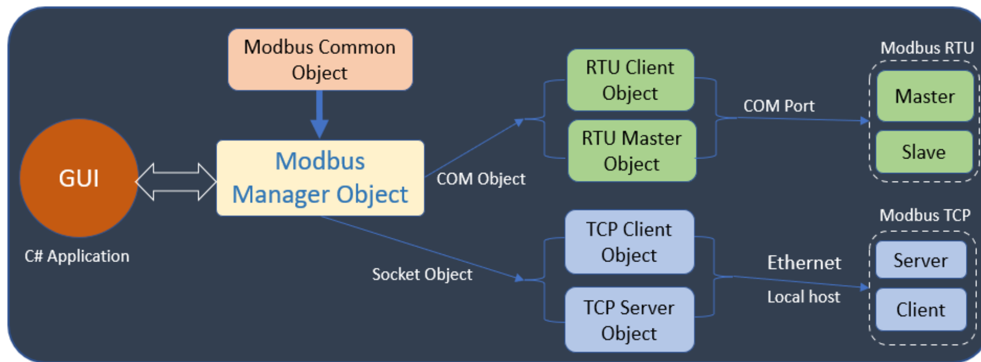

Fig. 1: Application Block Diagram.

Let us see what is going on behind the scene, which is depicted in figure 1. We create a Modbus Manager object which is responsible for managing all Modbus communication. All elements of Graphical User Interface (GUI) interact with Modbus Manager. The Modbus Common object is assisted to the Modbus manager providing some Modbus-related function. In the Modbus Manager, RTU Master/Client and TCP Server/Client objects are created. According to the Application Mode selected by the User, the request and response are handled by the specific channel. In our c# application, we integrated four types of popular functionalities in industrial automation (figure 2). Modbus RTU Master/Slave and Modbus TCP Server/Client mode. In the TCP mode, communication happens through a socket on an ethernet medium. The server always stays in an active listening mode, and the client initiates the contact. The client requested to send the required data providing function code, memory start address, number of a 16-bit register. In this mode, we do not care about CRC checks. The TCP/IP protocol takes care of packet health like integrity, packet corruption, etc.

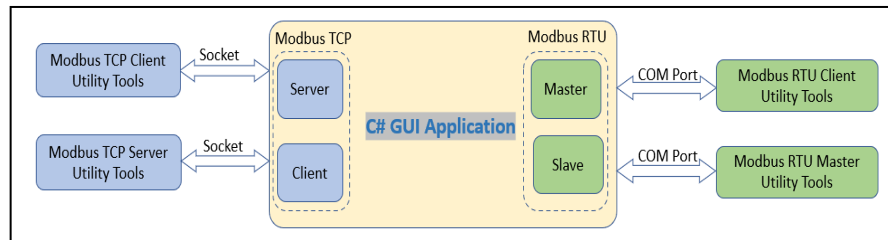

Fig. 2: Application Block Diagram.

Figure 3 depicts the Modbus TCP request packet structure. The Modbus TCP packet structure is on the left side, and on the right side, one request packet example is displayed. The index 0 and 1 are transaction identifiers. It is a 16-bit variable. When we send a request, take the variable, convert it into 2 bytes and push to buffer index 0 and 1. index 0 is LSB, and index 1 is MSB. Some applications are not swapped LSB and MSB. Index 2 and 3 are protocol identifiers by default 0x0000. The buffer index 4 and 5 are message lengths. The length is calculated from index 6 to the end, i.e., byte index 11. Buffer Index 6 byte is the unit identifier. This is the device address, which is connected to the communication network. Usually, the network is built by RS485 or ethernet framework. Every device has one unique id. Inherently Modbus

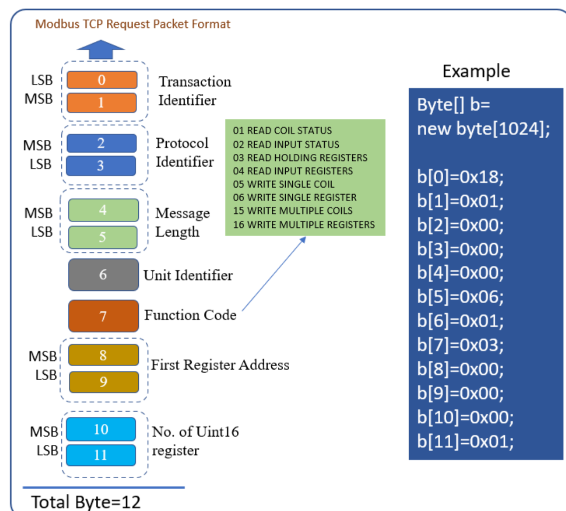

Fig. 3: Modbus TCP request Packet

communication is broadcast type. When the master requests some data, it reaches all the connected devices. The only device that replies matches with the unit identifier field in the received packet.
The byte index 7 is the Function code. There are several functions for different requests—the green box list the function code list. The byte index 8 and 9 is the first register address. in the device, the internal data structure is mapped with the memory address. This first register address is a 16-bit address. Converting from 16bit variable into two bytes, push into buffer Index 8 and 9. Index 10 and 11 are the number of 16-bit register requests. Figure 4.3 depicts the allocated coil or register number range associated with IO type.

Coil/Register Numbers	Data Address	IO Type	Access	Size(bit)
00001-09999	0000-9998 (0000-270E)	Output (Coils)	RW	1
10001-19999		Input (Contacts)	R	1
30001-39999		Input Register	R	16
40001-49999		Holding Register	RW	16

Fig. 4: Memory address range of Modbus register

The right side of figure 4 depicts one example of the request packet. Request for some data initiates by creating a byte array. Then push the data and send it through the socket. In the figure 4.2, Byte index 0 and 1 is 0x18, 0x01. It needs to swap the value. It is finally 0x0118. The decimal value is 280. Usually, It is increased sequentially, i.e., increment by 1. Byte index 2 and 3 is 0x0000 is protocol identifier. Index 4 and 5 is 0x006. It is byte count in the message buffer. This is the number of bytes present in the buffer that starts from index 6. The buffer index 6 is 0x01 which is device id. Theoretically, the device that may be present in the bus is 256. But in RS485 architecture, up to 31 devices can communicate well with the master. The index 7 is 0x03—the function code. Index 8 and 9 is 0x0000, which starts memory address. The buffer Index 10 and 11 is 0x0001, which is requested, 16-bit register count. Figure 5 depicts the Modbus TCP response packet. The transaction identifier field is the same value as the request packet. When the server device/application responds to the client, purse the request packet, save the transaction identifier field, and push the same data to the response packet buffer. The protocol identifier is the same, i.e., 0x0000. Then message length is the byte length, i.e., from index 6 to the last byte of the transmit buffer. The unit identifier and function code are the same as the request packet. The data byte count is placed into index 8. Up to index eight is fixed. The payload or data starts from index 9.

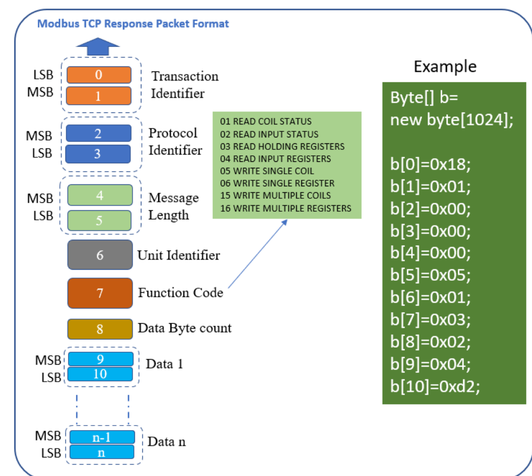

Fig. 5: Modbus TCP Response Packet

On the right side of figure 5, We see the example of the response packet. Before sending the response, we create a byte buffer. It can be any length. But need to assure the buffer length should not exceed the request data length. If it happens, we can reply by exception packet. Buffer index 0 and 1 is the transaction identifier. It is pushed from the request packet. Index 2 and 3 is the protocol identifier, by default 0x0000. Index 4 and 5 is the byte length present in the response packet that starts from index 6. Index 6 is 0x01, which is the device address. index 7 is 0x03 which is function code. Index 8 is 0x02 that is a number of data as a byte. The data byte starts from index 9. Index 9 and 10 is data in hex 0x04d2—decimal value 1234.

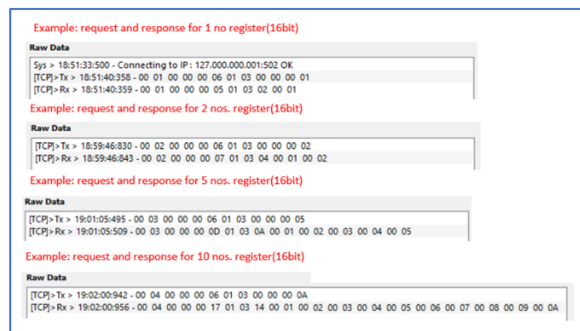

Fig. 6: Modbus TCP sample data Packet

Figure 6 depicts the sample packet of Modbus TCP, which is captured from the live system. The four data example has been given.

Figure 7 depicts the Modbus RTU request packet structure. The left side of the figure shows the packet structure, and on the right side, we provide an example of a packet. The slave device receives a pack of 8 bytes. The first index of byte buffer is slave address. Slaves should not process the packet further if the address does not match its address. Index 1 is function code. Index 2 and 3 is the first read register address. This is the first location from where we want to read the register content. It is a 16-bit address—the two-byte convert into a 16-bit value. Index 4 and 5 is the number of 16-bit data count. Index 6 and 7 is the CRC of the received packet. The CRC is calculated from index 0 to index 5. If the calculated CRC matches with the received CRC, we receive the packet. Else discard. On the right, we see an example. The index 0 is the device address. Index 1 is function code. Index 2 and 3 is the starting address of request data. Index 4 and 5 is the number of 16-bit data. Index 6 and 7 is the CRC of the received packet.

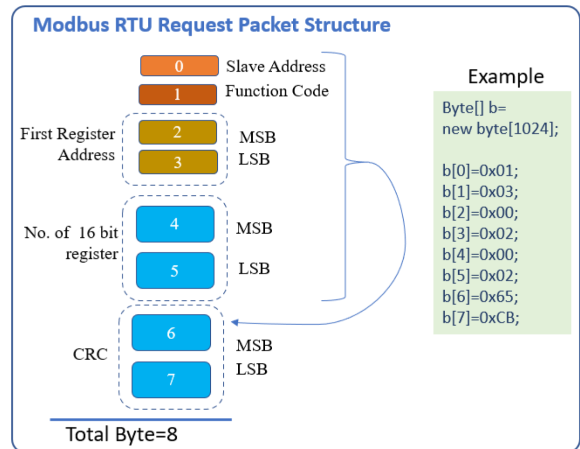

Fig. 7: Modbus RTU Request Packet

Figure 8 depicts the Modbus RTU response packet. The index 0 and 1 index are the same as the received packet. After receiving the packet from the master, it is pursued at the slave end and stored in the associated register. When we create the response packet, we push the data into transmit byte buffer. Slave address and function code we push into transmit buffer. Index 2 is the data count as a byte. From index 3, move the payload or data as a byte. Then we calculate the CRC and push it to the end of the packet.

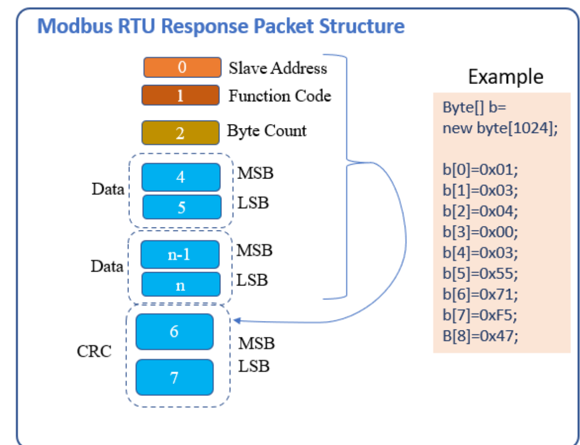

Fig. 8: Modbus RTU Response Packet

Modbus RTU Example

Modbus RTU 1 no. register request

```
[RTU]>Tx > 23:01:19:788 - 01 03 00 00 00 01 84 0A
[RTU]>Rx > 23:01:19:888 - 01 03 02 00 00 B8 44
```

Modbus RTU 2 nos. register request

```
[RTU]>Tx > 23:04:28:692 - 01 03 00 00 00 02 C4 0B
[RTU]>Rx > 23:04:28:751 - 01 03 04 00 00 00 02 7B F2
```

Modbus RTU 5 nos. 16bit register request-response example

```
[RTU]>Tx > 23:05:41:876 - 01 03 00 00 00 05 85 C9
[RTU]>Rx > 23:05:41:958 - 01 03 0A 00 00 00 02 00 03 00 04 00 05 C2 B4
```

Modbus RTU 10 nos. 16bit register request-response example

```
> 23:07:24:627 - 01 03 00 00 00 0A C5 CD
> 23:07:24:713 - 01 03 14 00 00 00 02 00 03 00 04 00 05 00 06 00 07 00 08 00 09 00 0A B2 EA
```

Fig.- 9: Modbus RTU Sample Data Packet

3 and 4 is the data 0x0003. Index 5 and 6 is second data 0x5571. Calculating CRC from 0 to 6, insert the CRC at 7 and 8 indexes. The CRC position is changed on the data count into the packet.

Figure 9 depicts the Modbus RTU sample packet which is received from the live system.

CRC Algorithm: Micro Controller Code

```
unsigned int Get_CRC(unsigned char* bfr, uint8_t u8length)
{
    unsigned char i, j;
    unsigned int temp, temp2, flag;

    temp = 0xFFFF;
    for (i = 0; i < u8length; i++)
    {
        temp = temp ^ *(bfr + i);
        for (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;
 return temp;
}
```

CRC Algorithm: C# Code

```
int Get_CRC(byte[] pkt, UInt16 length)
        {
            int i, j;
            int temp, temp2, flag;

            temp = 0xFFFF;
            for (i = 0; i < length; i++)
            {
                temp = temp ^ pkt[i];
                for (j = 1; j <= 8; j++)
                {
                    flag = temp & 0x0001;
                    temp >>= 1;
                    if (flag != 0)
                        temp ^= 0xA001;
                }
            }
            temp2 = temp >> 8;
            temp = (temp << 8) | temp2;
            temp &= 0xFFFF;
            return temp;
        }
```

Fig. 10: CRC algorithm for Microcontroller and C#

Figure 10 provides an applicable code for the CRC algorithm, which is running into our application. The two approaches are popular for CRC calculation. Lookup table and calculation-based approach. The lookup table-based approach is computing efficient and consumes some memory to store the table into the ROM. Where memory is not limited, we think lookup table-based approach. Another one is the calculation-based approach. It takes less memory but takes some CPU cycles to calculate the CRC value of the received packet. Where memory space is limited, we can think about the algorithm-based approach to process the CRC.

Function 01 Read Coils Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x01
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Quantity of Coils Hi	0x00
[5]	Quantity of Coils Lo	0x01
[6]	CRC Lo	FD
[7]	CRC Hi	CA

Function 01 Read Coils Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x01
[2]	Byte Count	0x02
[3]	Data Hi	0x00
[4]	Data Lo	0x00
[5]	CRC Lo	B9
[6]	CRC Hi	FC

Function 02 Read Inputs Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x02
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Quantity of Coils Hi	0x00
[5]	Quantity of Coils Lo	0x01
[6]	CRC Lo	B9
[7]	CRC Hi	CA

Function 02 Read Inputs Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x02
[2]	Byte Count	0x02
[3]	Data Hi	0x00
[4]	Data Lo	0x00
[5]	CRC Lo	B9
[6]	CRC Hi	B8

Function 03 Read Holding Register Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x03
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0x00
[5]	16bit reg. count Lo	0x01
[6]	CRC Lo	84
[7]	CRC Hi	0A

Function 03 Read Holding Register Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x03
[2]	Byte Count	0x02
[3]	Data Hi	0x00
[4]	Data Lo	0x01
[5]	CRC Lo	79
[6]	CRC Hi	84

Function 04 Read Input Registers Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x04
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0x00
[5]	16bit reg. count Lo	0x01
[6]	CRC Lo	0x31
[7]	CRC Hi	0xCA

Function 04 Read Input Registers Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x04
[2]	Byte Count	0x02
[3]	Data Hi	0x00
[4]	Data Lo	0x01
[5]	CRC Lo	0x78
[6]	CRC Hi	0xF0

Function 0x05 Write Single Coil Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x05
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Data Hi	0xFF
[5]	Data Lo	0x00
[6]	CRC Lo	8C
[7]	CRC Hi	3A

Function 0x05 Write Single Coil Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x05
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Data Hi	0xFF
[5]	Data Lo	0x00
[6]	CRC Hi	0x8C
[7]	CRC Lo	0x3A

Function 0x06 Write Single Register Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x06
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Data Hi	0xFF
[5]	Data Lo	0x00
[6]	CRC Lo	0xC8
[7]	CRC Hi	0x3A

Function 0x06 Write Single Register Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x06
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	Data Hi	0xFF
[5]	Data Lo	0x00
[6]	CRC Hi	0xC8
[7]	CRC Lo	0x3A

Function 0x0F Write Multiple Coils Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x0F
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0xFF
[5]	16bit reg. count Lo	0x00
[6]	Byte Count	0x02
[7]	Data Hi	0x02
[8]	Data Lo	0x02
[9]	CRC Lo	0x72
[10]	CRC Hi	0x55

Function 0x0F Write Multiple Coils Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x0F
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0xFF
[5]	16bit reg. count Lo	0x00
[6]	CRC Hi	0x14
[7]	CRC Lo	0x3B

Function 0x10 Write Multiple Register Request Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x10
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0x00
[5]	16bit reg. count Lo	0x01
[6]	Byte Count	0x02
[7]	Data Hi	0x02
[8]	Data Lo	0x02
[9]	CRC Lo	0x26
[10]	CRC Hi	0xF1

Function 0x10 Write Multiple Register Response Packets

Position In Array	Field Name	Example (in Hex)
[0]	Slave Address	0x01
[1]	Function	0x10
[2]	Starting Address Hi	0x00
[3]	Starting Address Lo	0x00
[4]	16bit reg. count Hi	0x00
[5]	16bit reg. count Lo	0x02
[6]	CRC Hi	0x41
[7]	CRC Lo	0xC8

Fig. 11: Function Code Example

Figure 11 depicts several function code practical examples. When we prepare request and response packets, we need to insert the proper function code. Every function code is mapped with a specific memory address, so improper function code gets data from the wrong place. The critical application may cause a severe issue for operation. Need CRC checking to eliminate corrupt packet processes.

5. EXPERIMENT :

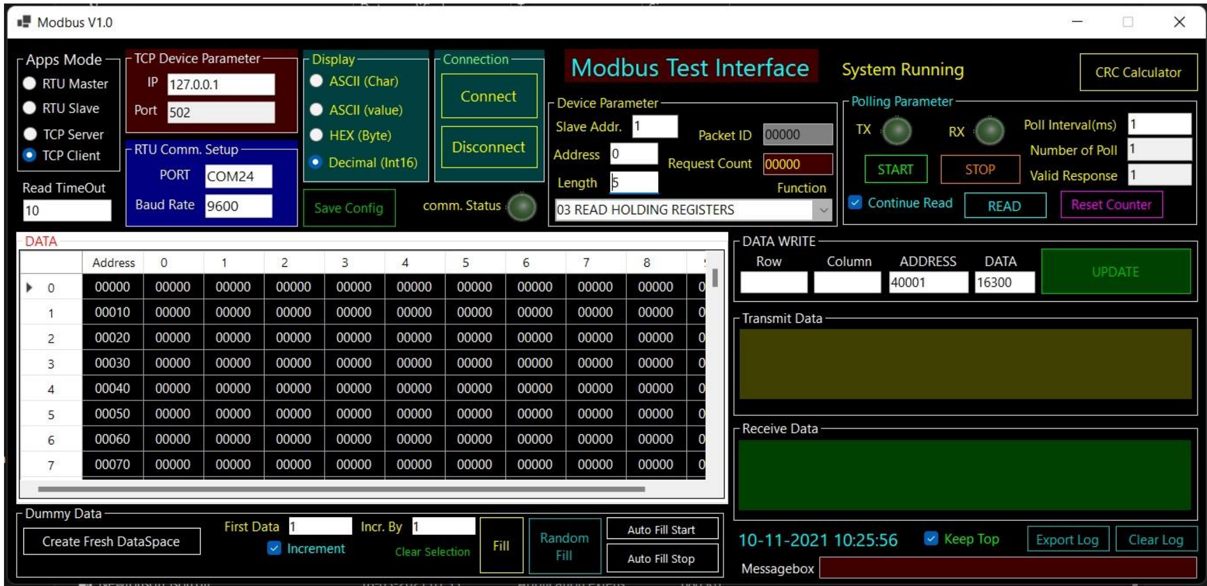

Fig. 12: Modbus Test Interface

Figure 12 depicts our Modbus interface GUI. Before doing some experiments, Let us introduce this interface which we have already developed.

Apps Mode: We can select the application mode from the top-left group box. We can choose RTU or TCP mode. In RTU mode, we can select master or client. Our software can be a master or client as well as a slave or server. Most of the utility software, both feature is not available. Our software is not a commercial product. So, we included this feature for our research purpose. Using this feature, we can run two application instances on the same PC with a different configuration. To run this feature on the same computer, we need two external USB-Serial ports. One Port will be connected with the master application, and another will be associated with the client application. In figure 13, using USB Hub, we combine two USB to serial converter modules. One module TX connected with another module RX and vice versa. The ground is common for both modules. We also have another option like a virtual comm port. Using a virtually comm port, internally be connected between two ports. If we select the RTU Master or TCP client radio button, The application will poll or request slave or server device data. The received data is populated into a data grid. For RTU Slave or TCP Server mode, our application can respond to the master request. When the master requests some data, it receives it and responds. The data will be sent from Datagrid. For our experiment, we can change the Datagrid value also. The various filling option is available below the Datagrid.

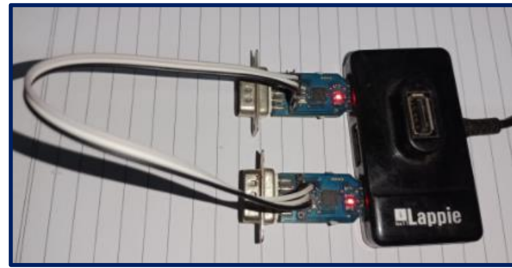

Fig. 13: Master-Slave using 2 Port

TCP Device Parameter: The TCP device parameters are required in TCP client mode. Before connecting with the remote Modbus TCP server, we need to set the remote device IP address and default Modbus port 502. If the local and remote apps are on the same computer, the client application IP address will be 127.0.0.1, i.e., localhost.

RTU Comm Setup: for Modbus RTU mode, we need to enter COM port and baud rate. The assigned port name is available inside the windows device manager. For the baud rate, we need to study the device specification or user manual.

Connection: After TCP device parameter setup or RTU COM. Setup, need to press connect. If the communication is established, the comm. Status LED will turn green. If not, it turns into a red. The disconnect button is used to discontinue the communication.

Display: The display section is used to set the display data format. Four types of format are available ASCII char or ASCII value or HEX, or Int16 decimal format. If it is not sufficient, we can add more display formats. C# programming knowledge is mandatory to change something.

Save Config: after the application starts, we change some configurations to run our application. Pressing The save config button, we can save configuration data. The configuration data is also automatically saved upon closing the application.

Device Parmeter: Some device-related settings are available in this section. The slave address field is the device id. In RTU master or TCP client mode, insert this value as unit id to the request packet. When apps are set as Modbus clients, this field is used to compare with received packet id. If the id matches, the packet is processed further. Else the packet is discarded. The address text field is used as the start read address when apps run in RTU master or TCP client mode. The length textbox value is used as 16 bits register in RTU master or TCP client mode. The function Combobox is used to select the function code. The chosen function code is inserted into the transmitting packet when apps run on RTU master or TCP client mode. The packet ID field is read-only. When apps run in TCP client mode, this textbox value is used as a transaction identifier value. After one transaction, this value is incremented by 1. The request count field is also another read-only textbox. When apps run in RTU slave or TCP server mode, it indicates how much time requests for data from RTU master or TCP client.

Polling Parameter: under this section, some data polling-related parameters are present. After connection, we need to start a timer that will check every millisecond of any request from outside the application. If any request is available, the application will execute send replies back to the outside of the apps. So after connecting, the following essential activities are to start the polling. The start button is for start polling and any time to stop polling need to press the stop button. We can set polling intervals. This interval is used when this application is used to fetch data from outside of the apps. The number of poll fields is used to see how much time send data request and valid response field indicates that several correct responses get from an external application that may be Modbus RTU client or Modbus TCP server in Modbus TCP mode. We can reset the counter any time by pressing the reset counter

button. The data exchange between the application and device RX and TX indicates receiving and transmitting. The READ button is used to read only once and non-stop read if we checked on continue read. The application will automatically read the data from the external device within the polling interval value.

Data Write: if we want to write any data to the device from our application, this section will help us. It is used in a master mode in Modbus RTU or TCP client mode. First, select the data grid cell in which data we want to write. The corresponding row and column will display in the text box. The specific address location will correspond. Now we can change the data into the textbox, or it will take from grid data by default. Then press update. It will, by default, send the data to the client using function code 06 that is to write a single register.

CRC Calculator: Sometimes, we need to calculate CRC. We take the help of an online CRC calculator. Here we provide a CRC calculator for our research work. When we need to calculate CRC on a given array, we need to press the button. A separate window will open. In the data grid, we need to fill the data from the top row. After all, entries need to press the CALCULATE CRC button. The result will be displayed in the CRC box. The result is swapped. The LSB first, then the MSB. For repetition, we need to clear all entries. So press the ROW Clear button to remove the rows. Then again, we can start. Figure 14 depicts the CRC calculator window.

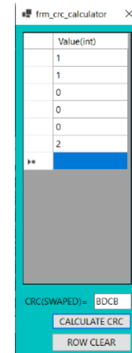

Fig. 14: CRC Calculator

Transmit Data: Whatever data we send to the outside of the application. It will show here. It is used for debugging purposes also. We can watch these windows and can see the packet is sent to the outside of the application is correct or not. It is too practical for first-time algorithm development. To clear the log, the clear log button is provided.

Receive Data: The receive data box is used to audit the data. Whatever the data is received is shown here. It is used to show the data sent from outside of the apps. This box can be cleared through the Clear Log button. The Keep Top checkbox is used to keep the last data on top. If it is not required, it should be unchecked.

Message box: This textbox shows various messages. Generally, any application shows a different message through the message box. It is an unpleasant experience. To display the content silently, we provide the message box to see the notification without interruption.

Data Grid: The data grid displays incoming data when the application is in RTU slave or TCP client mode. This data grid is also used when we send the data in RTU master or TCP server mode. Click on the associated Datagrid cell, update the cell value, and press enter to change the data.

Dummy Data: The below of the Datagrid is the dummy data manipulation section. We can see the Dummy data section. From the left side, the first button is to clear the data grid row selection. The Random fill button is used to fill Datagrid with random Uint16 data. It is used basically for dummy data sent to the device or outside of the application. The fill button is used to fill the data grid with conditioned data. The first data field needs to fill first, which data would be the first, put here. Then if the following data will increment, checked on-increment. And then need the following information is incremented by. This field data will automatically increment. Then finally, press the fill button. When we press the “Auto Fill start” button, The Datagrid data is filled with random data every one-second interval. To stop this, press “Auto Fill Stop.”

Now we can do some practical experiments with the Modbus protocol through Modbus interface software. We need to download our project From the GitHub repository. To open the project file, we need to install the Microsoft visual studio community edition. All Link is available in recommend section. After installing the visual studio IDE, we can open the project. Under the build menu, clean and build the solution. After successfully building, from debug menu, we can press start debugging. If our utility software is open, it looks like figure 5.1. now start our communication—four types of tests we can execute through it. The procedure is explained below.

RTU Master Mode: if we have any slave device and want to fetch data from that device, select RTU Master.

1. Open our application. From the “App Mode” group box (Top-Left), click the “RTU Master” radio button.

2. Under the "RTU Comm. Setup" group box Enter PORT and Baud rate. The assigned PORT number can be found from the device manager and the baud rate from the device specification.
3. Press the "Connect" button in the "Connection" group box if the comm connects the application. Object/device, "comm. Status" LED will turn green. Else will glow red.
4. In The "Device parameter," group box enters the "Slave Addr.", "Address," and "Length" text fields. Form function combo box, select desired function.
5. From the "polling Parameter" group box, press the "START" button. Check the "Continue Read" check box.
6. That's it. The received data will display in the Data grid.
7. Whatever configurations are entered are saved upon closing the application by the top-right close button, and the "Save Config" button can do the same action. When the application reopens, all configurations will be retrieved into their respective field.

RTU Slave Mode: if we have any Modbus Salve device/application and want to provide data to the device/application, from App Mode, select "RTU Slave" and need to configure it as below.

1. Under "RTU Comm. Setup" Enter PORT and Baud rate. PORT can be found from the device manager and baud rate from device specification.
2. Press the "Connect" button in the connection group box if the application is connected with the device/application, comm. Status LED will turn green else red.
3. The "Device parameter" group box only enters the "Slave Addr."
4. From the "polling Parameter" group box, press the "START" button.

That's it. The master device/application will get the data from the data grid. We can change data grid data. If we press the "**Auto Fill start**" button, The grid data will change randomly with a one-second interval. The auto-fill can be stopped by pressing "**Auto Fill Stop**." Button.

TCP Server Mode: if we have any Modbus TCP client device/application, we can provide data to the device/application. We need to follow the below steps.

1. From Apps Mode, select "TCP server."
2. Press the "Connect" button in the connection group box if the application is connected with the device, comm. Status led will turn green. Else turn red.
3. The "Device parameter" group box enters the "Slave Addr."
4. From the "polling Parameter" group box, press the "START" button.

That's it. If data is transmitted and received, TX and RX LED will glow. We can change the "DATA" provided to the client as discussed in the RTU Slave Mode.

TCP client Mode: if we have any Modbus TCP server/application and want to fetch data to our application, we need to follow the below steps.

1. From App Mode, select "TCP client."
2. In "TCP Device Parameter," enter the IP address of the server application/ device. The same system will be "127.0.0.1". The Port number is that of the server Communicates.
3. Press the "Connect" button.
4. In the "Device parameter," fill "**Slave address**.", "**Address,**" and "**Length**" fields.
5. From the "polling Parameter" group box, press the "START" button.

That's it. The reading data will display in the Datagrid.

6. RECOMMENDATIONS:

- The complete source code is available from https://github.com/sudipchakraborty/Demonstration-of-Robot-Communication-On-Modbus-Using-C-Sharp
- For Microsoft visual studio 2019 community edition, download the link https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=16

- For this research work, we used Modbus utility software to test our application. We can download it from https://sourceforge.net/projects/qmodmaster/files/latest/download
- The above research work is reference only. Taking this project, we need to customize it. We wanted to provide some ideas to the new researcher on how Modbus works, communicate through a serial port, TCP/IP socket communication.
- We can run more than one application instance with the different configurations to understand Modbus better.
- This application is not entirely bug-free. We debugged what we got at experiment time. May persists several bugs, which can be debugged under more tests.
- We developed core functions only. The researcher can add some additional functionality for better usability.
  - Using our Auto Fill start/stop button, we can change the random value of the entire grid. Sometimes may need to change incrementally or randomly to specific data. It is required for sensor simulation.
  - In the real scenario, noise is imposed on actual sensor data. We can simulate here by changing the grid value by noise value generated by some algorithm or using the lookup table.
  - For accessible audit features, we can add a data search option.
  - Background color may be changed change for different app modes to identify at a glance.
  - To make this robust, Error Handling may be reviewed.
- Any application-related suggestions are appreciated.

7. CONCLUSION :

The Modbus is a popular protocol in the industrial automation domain. Our robot researcher may need to integrate the protocol to communicate with Modbus-enabled devices. On the web, most of the available documents are not practical oriented. Our new robot researchers face challenges in integrating the Modbus into their project. Because lots of issues like the CRC algorithm must be accurate, packet parsing and response packet should be precisely in order, etc. Otherwise, the interface throws an error. In this paper, we provide some practical-oriented documentation so they incorporate the Modbus easily. We provided interface software also, so initially, they could start experiments without writing any code.

REFERENCES :

[1] Tamboli, S. Rawal, M. Thoraiet, R. and Agashe, S. (2015). Implementation of Modbus RTU and Modbus TCP communication using Siemens S7-1200 PLC for a batch process. International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM), 258-263.
Google Scholar↗ CrossRef/DOI↗

[2] Jaloudi, S. (2019). Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study. Future Internet 2019, 11, 66.
Google Scholar↗ CrossRef/DOI↗

[3] Ebenhofer, G., Bauer, H., Plasch, M., Zambal, S., Akkaladevi, S. C. and Pichler, A. (2013). A system integration approach for service-oriented robotics. IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA), 2013, 1-8.
Google Scholar↗ CrossRef/DOI↗

[4] Lin H. and Hwang, Y. (2019). Integration of Robot and IIoT over the OPC Unified Architecture. (2019). International Automatic Control Conference (CACS), 1-6.
Google Scholar↗ CrossRef/DOI↗

[5] Tarek Al-Geddawy, (2020). A Digital Twin Creation Method for an Opensource Low-cost Changeable Learning Factory. *Procedia Manufacturing,* 5(1), 1799-1805.
Google Scholar↗ CrossRef/DOI↗

[6] Hemmatpour, M., Ghazivakili, M., Montrucchio, B. and Rebaudengo, M. (2017). DIIG: A Distributed Industrial IoT Gateway. IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 755-759.

Google Scholar↗ CrossRef/DOI↗

[7] Qingdao Huang, Qianzhong She and Xiaofeng Lin, (2010). Adaptive fuzzy PID temperature control system based on OPC and Modbus/TCP protocol, 2nd International Asia Conference on Informatics in Control, Automation and Robotics (CAR 2010), 2010, 238-241.
Google Scholar↗ CrossRef/DOI↗

[8] Maldonado V., Gamboa S., Trujillo M.F., Rodas A. (2021) Development of an Industrial Data Server for Modbus TCP Protocol. In: Botto-Tobar M., S. Gómez O., Rosero Miranda R., Díaz Cadena A. (eds) Advances in Emerging Trends and Technologies. ICAETT 2020. Advances in Intelligent Systems and Computing, vol 1302. Springer, Cham.
Google Scholar↗ CrossRef/DOI↗

[9] Xiang, X., Liu, H., Yu, C. and Niu, Z. (2016). A newly developed hybrid underwater robotic vehicle (HURV): Communication design and tests. OCEANS 2016 - Shanghai, 2016, 1-6.
Google Scholar↗ CrossRef/DOI↗

[10] Hewei, G. Xingdong, L. and Yangwei, W. (2018). Design of Control System for Educational Robot with Six-Degree Freedom. 11th International Workshop on Human-Friendly Robotics (HFR), 2018, 85-90.
Google Scholar↗ CrossRef/DOI↗

[11] Jingran, W., Mingzhe, L., Aidong, X., Bo, H., Xiaojia, H. and Xiufang, Z. (2020). Research and Implementation of Secure Industrial Communication Protocols. IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS), 2020, 314-317.
Google Scholar↗ CrossRef/DOI↗

****