Document Type: Tutorial
NI Supported: Yes
Publish Date: Feb 18, 2009

Introduction to MODBUS

Overview

MODBUS is an application-layer protocol based on a client/server or request/reply architecture. It was published by Modicon and 1979 and is primarily used in industrial applications. The following tutorial outlines the high level functionality of the MODBUS application layer with emphasis on the specifications for a serial implementation and TCP/IP specification. For more details, please see the complete MODBUS specifications on www.modbus.org.

Table of Contents

MODBUS Protocol

Introduction

The MODBUS protocol follows a client/server (master/slave) architecture where a client will request data from the server. The client can also ask the server to perform some action. The client initiates a process by sending a function code that represents the type of transaction to perform. The transaction performed by the MODBUS protocol defines the process a controller uses to request access to another device, how it will respond to requests from other devices, and how errors will be detected and reported. The MODBUS protocol establishes a common format for the layout and contents of message fields.

During communications on a MODBUS network, the protocol determines how each controller will know its device address, recognize a message addressed to it, determine the kind of action to be taken, and extract any data or other information contained in the message.

Controllers communicate using a master/slave technique where only one device, the master, can initiate transactions or queries. The other devices, slaves, respond by supplying the requested data to the master or by taking the action requested in the query. Typical master devices include host processors and programming panels. Typical slaves include programmable controllers.

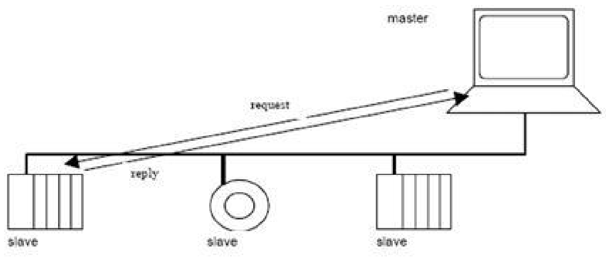


Figure 1: Basic MODBUS Network[1]

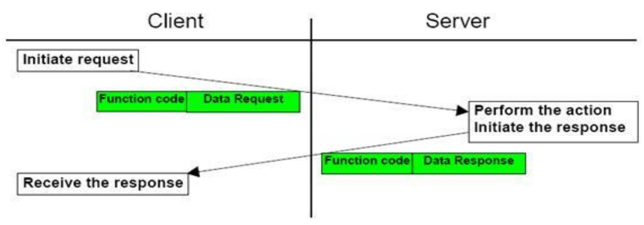


Figure 2: Basic MODBUS Transaction[1]

The messages exchanged between the client and the server is called frames. There are two types of MODBUS frames: Protocol Data Unit (PDU) and Application Data Unit (ADU). The PDU frames contain a function code followed by data. The function code represents the action to perform and the data represents the information to be used for this action. ADU frames add a little more complexity with an additional address part. ADU frames also provide some error checking. Both the ADU and PDU frames follow Big-Endian encoding.

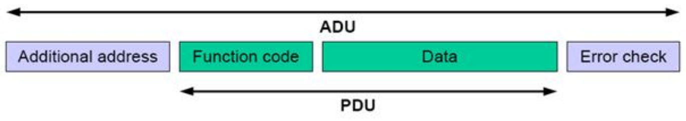
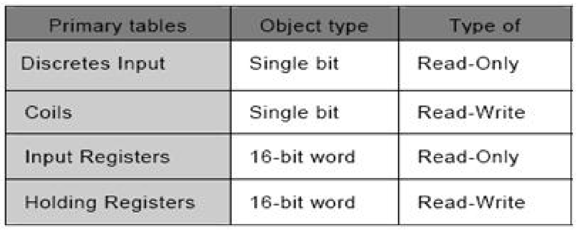


Figure 3: MODBUS Frame[1]

MODBUS transactions always perform a set of actions by reading or writing to a set of four data types. Table 1 describes the four data formats used by the MODBUS application layer.



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

Table 3: MODBUS Data Types[1]

The **Discrete Inputs** represent a single bit (Boolean) which can only be read. In other words, the client can only perform a read action on the discrete inputs. The same holds for the **Input Registers**. The client can only read the server's Input Registers. The difference between the Discrete Inputs and the Input Registers is that the Input Registers represent 16 bits while the Discrete Inputs are only a single bit. The **Coils** also represent a Boolean data type which can be read and written from the client. The **Holding Registers** represent a 16 bit word that can be read and written to.

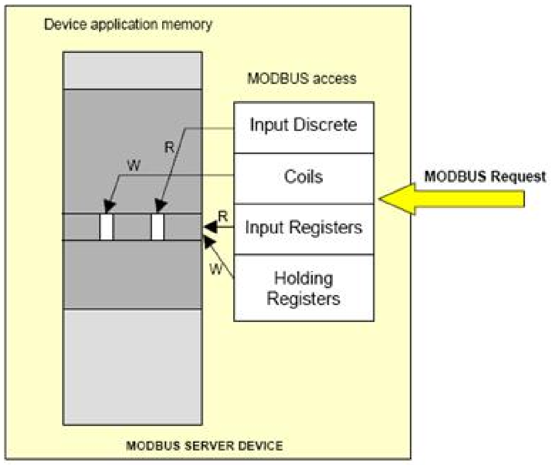
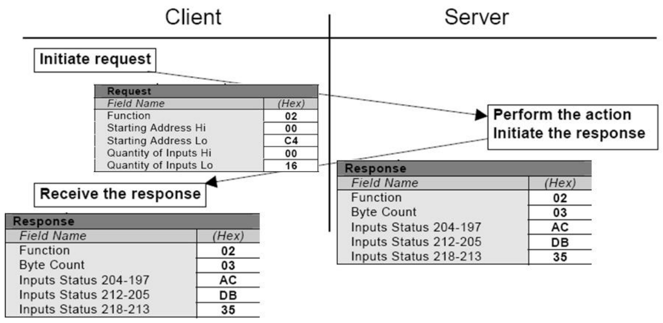


Figure 4: MODBUS Transaction with Data Types[1]

The Complete MODBUS Transaction

As mentioned before, the type of action that the server performs is defined by a function code. For example, if the client wants to reads a certain Discrete Input, it will send a function code of 0x02 followed by the address of the desired Discrete Input. The server will read the 0x02 and will know that the client wants a Discrete Input. The server will retrieve the Discrete Input from the given address and reply back to the client.



Request	
Field Name	*(Hex)*
Function	02
Starting Address Hi	00
Starting Address Lo	C4
Quantity of Inputs Hi	00
Quantity of Inputs Lo	16

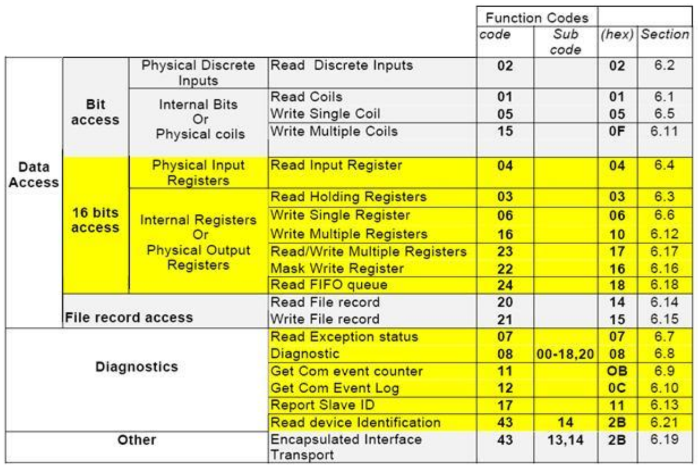
Response	
Field Name	*(Hex)*
Function	02
Byte Count	03
Inputs Status 204-197	AC
Inputs Status 212-205	DB
Inputs Status 218-213	35

Response	
Field Name	*(Hex)*
Function	02
Byte Count	03
Inputs Status 204-197	AC
Inputs Status 212-205	DB
Inputs Status 218-213	35

Figure 5: Complete MODBUS Transaction[1]

More on Function Codes

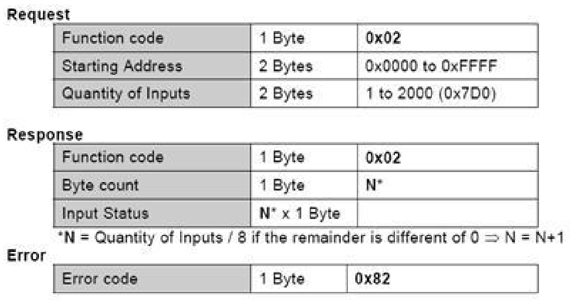
There are three main types of function codes: Public, User Defined, and Reserved. Since the public function codes are validated, publicly documented, and have available conformance tests by the MODBUS-IDA.org community, most MODBUS devices implement them. Each of the public function codes are associated with a well defined function. A brief overview of the public function codes is presented in Table 2. More information on public function codes can be found at www.modbus.org.



				Function Codes			
				code	Sub code	(hex)	Section
Data Access	Bit access	Physical Discrete Inputs	Read Discrete Inputs	02		02	6.2
		Internal Bits Or Physical coils	Read Coils	01		01	6.1
			Write Single Coil	05		05	6.5
			Write Multiple Coils	15		0F	6.11
	16 bits access	Physical Input Registers	Read Input Register	04		04	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	03		03	6.3
			Write Single Register	06		06	6.6
			Write Multiple Registers	16		10	6.12
			Read/Write Multiple Registers	23		17	6.17
			Mask Write Register	22		16	6.16
			Read FIFO queue	24		18	6.18
	File record access		Read File record	20		14	6.14
			Write File record	21		15	6.15
Diagnostics			Read Exception status	07		07	6.7
			Diagnostic	08	00-18,20	08	6.8
			Get Com event counter	11		OB	6.9
			Get Com Event Log	12		0C	6.10
			Report Slave ID	17		11	6.13
			Read device Identification	43	14	2B	6.21
Other			Encapsulated Interface Transport	43	13,14	2B	6.19

Table 2: Public Function Codes[1]

In the previous example, we used function code 0x02 to read the Discrete Inputs. We can look at this function code in more detail, since it is a public function code.



Request

Function code	1 Byte	0x02
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Inputs	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	0x02
Byte count	1 Byte	N*
Input Status	N* x 1 Byte	

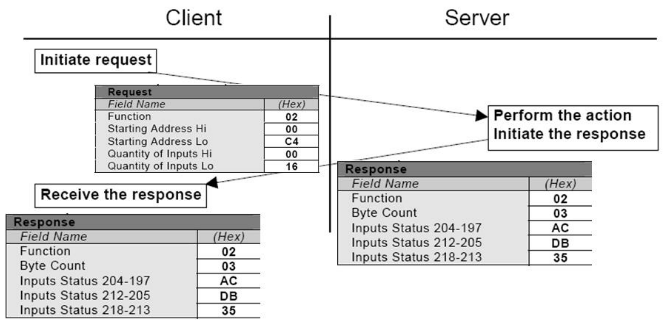
*N = Quantity of Inputs / 8 if the remainder is different of 0 ⇒ N = N+1

Error

Error code	1 Byte	0x82

Figure 6: Detailed Function Code Example[1]

As seen in Figure 6, the function code must be followed by 2 bytes for the starting address and 2 bytes for the number of inputs the client requires. The server must respond with the function code, followed by the 1 byte representing the number of bytes sent and then the Discrete Input values. The complete transaction is shown below.



Request

Field Name	(Hex)
Function	02
Starting Address Hi	00
Starting Address Lo	C4
Quantity of Inputs Hi	00
Quantity of Inputs Lo	16

Response

Field Name	(Hex)
Function	02
Byte Count	03
Inputs Status 204-197	AC
Inputs Status 212-205	DB
Inputs Status 218-213	35

Response

Field Name	(Hex)
Function	02
Byte Count	03
Inputs Status 204-197	AC
Inputs Status 212-205	DB
Inputs Status 218-213	35

Figure 7: Complete MODBUS Transaction[1]

In contrast, the user defined codes are unique for each MODBUS device. They are usually tied with a special set of functions which are only available for the specific device. These will be detailed in the device manufacturer's manual.

Hardware Implementations

Serial Implementation

There are two serial modes that the MODBUS application layer can follow: RTU and ASCII. In RTU, the data is represented in Binary format, whereas the ASCII mode represents the data such that it is human readable. Figure 8 and 9 demonstrate the difference between these two modes.

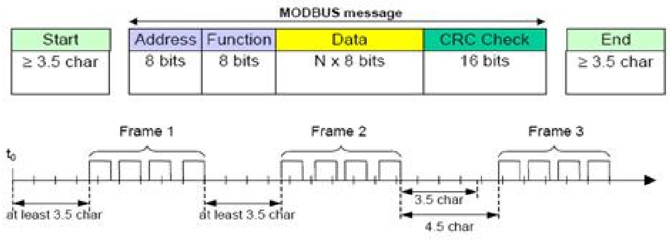
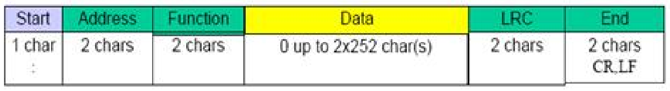


Figure 8: MODBUS RTU Serial Frame[1]



Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Figure 8: MODBUS ASCII Serial Frame[1]

The most common serial protocols used with MODBUS are RS-232 and RS-485. For more details on these protocols, please see this link.

TCP Implementation

As in many TCP applications, the first requirement is to establish a connection between the client and the server. In MODBUS, the server will be listening on port 502 and the connection establishment follows the TCP/IP protocol. When connection has been established, the client can build a request for the server. The request contains a PDU (MODBUS frame described above) followed by a MPAB header, as shown in Figure 10. Table 3 represents a template for the MPAB header.

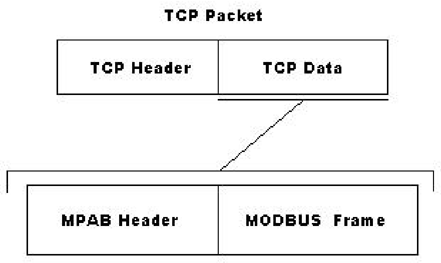
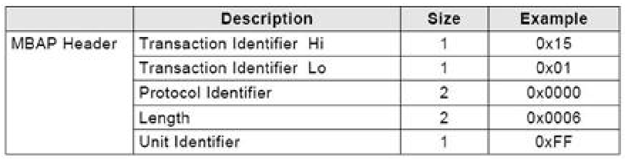


Figure 9: MODBUS TCP Frame



	Description	Size	Example
MBAP Header	Transaction Identifier Hi	1	0x15
	Transaction Identifier Lo	1	0x01
	Protocol Identifier	2	0x0000
	Length	2	0x0006
	Unit Identifier	1	0xFF

Table 3: MBAP Header[1]

The Transaction Identifier can be like a "TCP Sequence Number" used to keep track of which MODBUS transaction the packet is associated with. This is important because, in MODBUS TCP, the server can handle many requests at the same time. This is not possible in MODBUS Serial.

The Unit identifier is typically used to address the MODBUS slave. When using MODBUS TCP, the address of the slave is its IP address and the Unit Identifier in the MBAP header is not used. Figure 10 demonstrates a complete MODBUS TCP transaction.

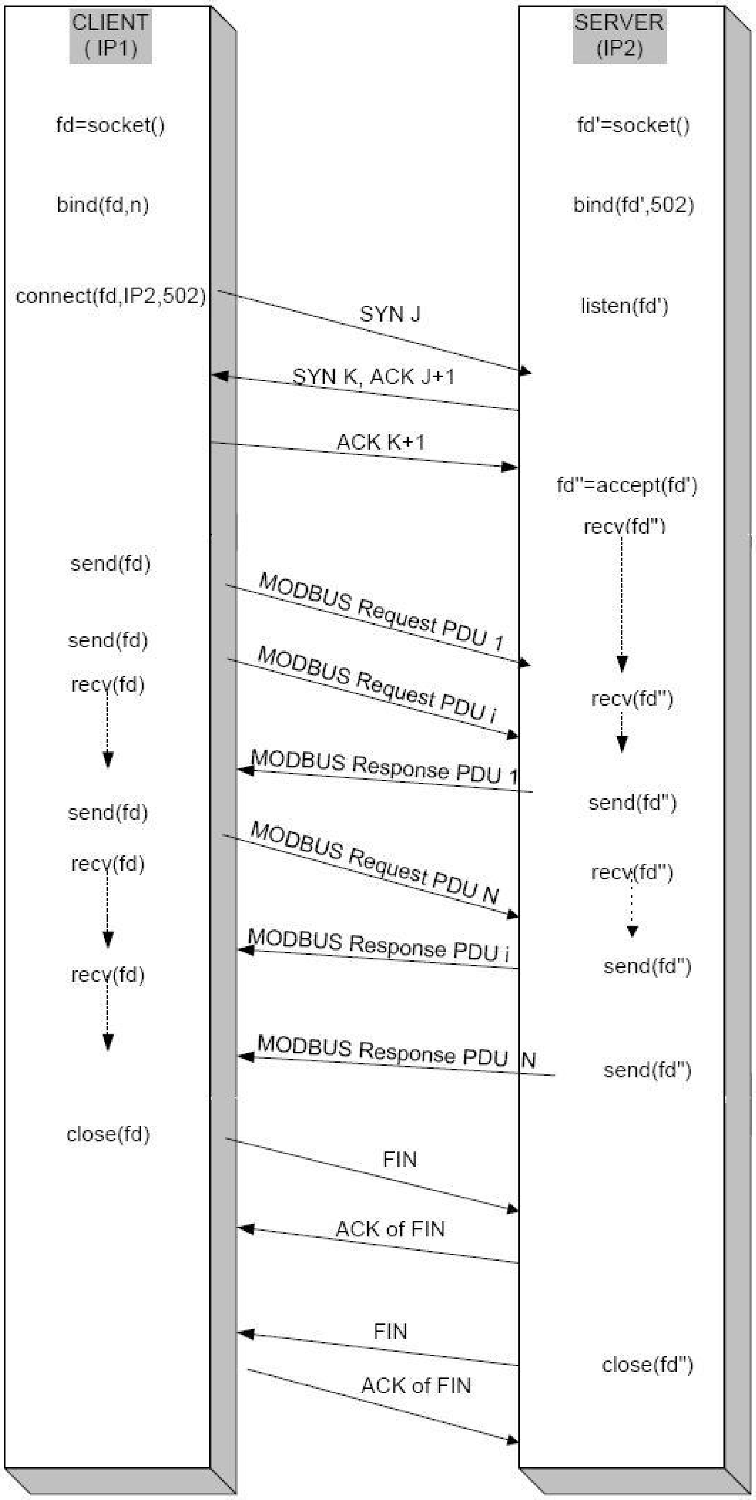


Figure 10: Complete MODBUS TCP Transaction[1]

LabVIEW Libraries

You can download free MODBUS libraries for LabVIEW. A great point to start for your application is to look at the examples supplied with the libraries. There are examples for master (client) and slaves (server) for both the serial and TCP implementation of the MODBUS protocol.

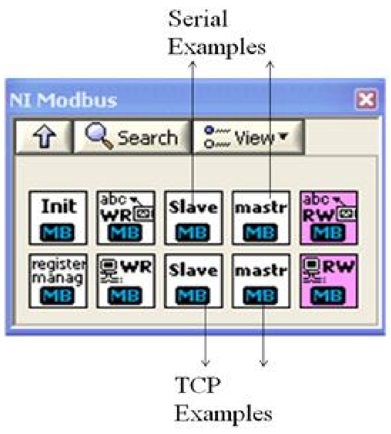


Figure 11: LabVIEW MODBUS Libraries

MODBUS Serial Master

The first operation to be done in MODBUS serial master is to open a VISA session and initialize the COM port with all the proper configuration parameters (baud rate, start bit, stop bit, etc).

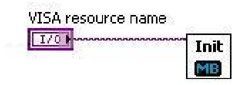


Figure 12: VISA Open and Configure Session to Serial Port

As explained in the previous paragraphs, the master (client) usually queries the slave (server) for some data. In the LabVIEW libraries, this is represented by a while loop which continuously queries the slave for the Discrete Inputs, Coils, Input Registers and Holding Registers.

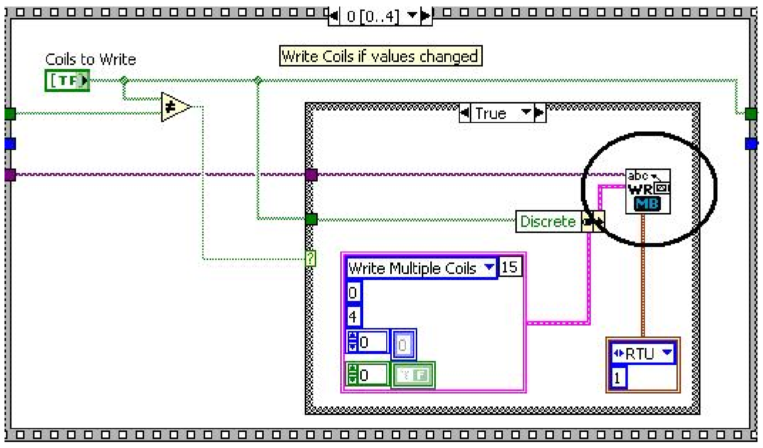


Figure 13: Master Main Loop

If we look deeper into the MB Serial Master Query.vi, we see four main VIs used in sequence. The first formats the data into a MODBUS frame. This frame is then written to the serial port using a VISA Write VI. The master then expects a response from the slave, so a VISA Read VI is called to read the bytes at the serial port. This information is then reformatted to be handled and displayed in LabVIEW.

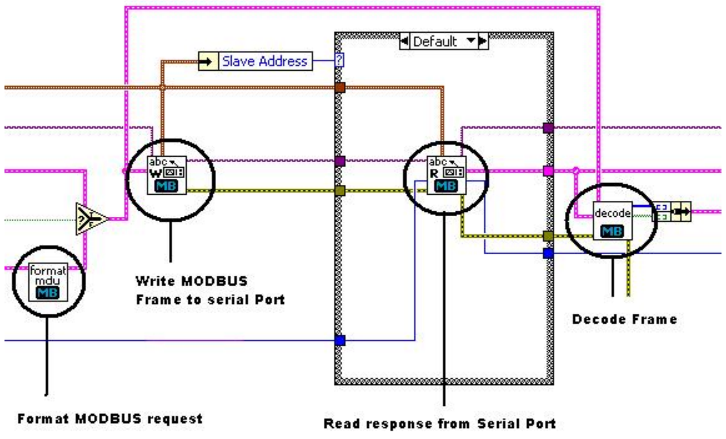


Figure 14: Master Query

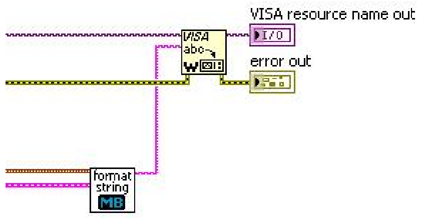


Figure 15: VISA Write

On the slave side, we also need to continuously monitor the serial port for requests. In this case, we use a timed loop to ensure that the serial port is read at specific intervals.

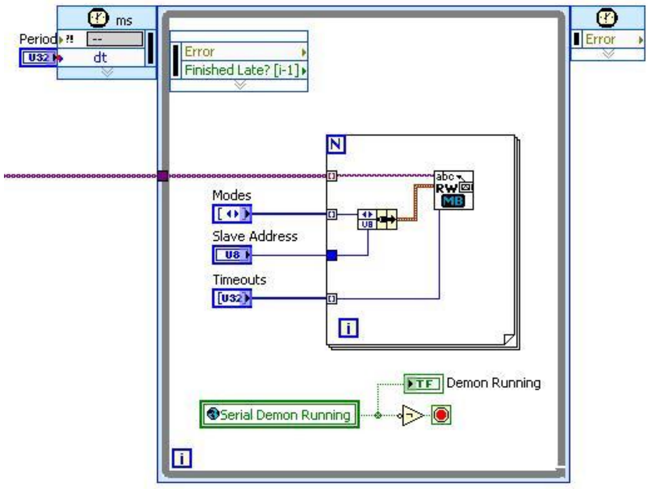


Figure 16: Slave Main Loop

MODBUS TCP

Similarly to opening a VISA session to a serial port, the first operation done by the master (client) is to connect to the slave (server).

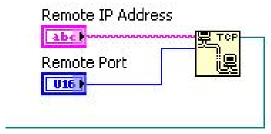


Figure 17: TCP Open Connection

The next action taken from the master is to send a query to the slave. As in the serial case, we need to organize the data to conform with the MODBUS TCP frame specification.

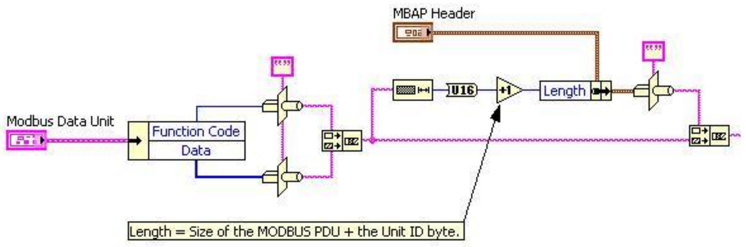


Figure 18: Format TCP MODBUS Request

In contrast to a VISA Write, we use a TCP Send.vi to send the MODBUS frame to the TCP connection.

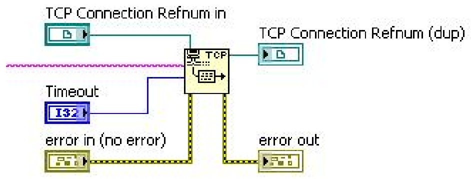


Figure 19: TCP Send

This is also followed by a TCP Read.vi as the master expects a frame back from the slave.

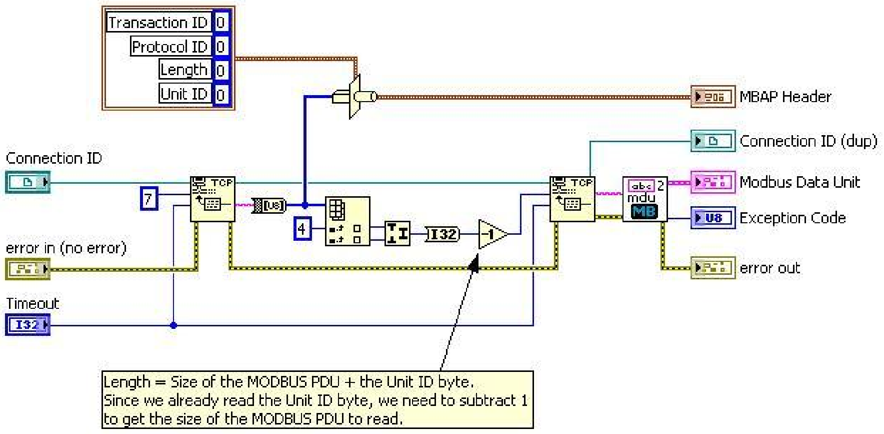


Figure 20: TCP Receive

Since the MODBUS TCP slave can communicate with more than one device at any point in time, it needs to continuously monitor if other masters are trying to establish a connection.

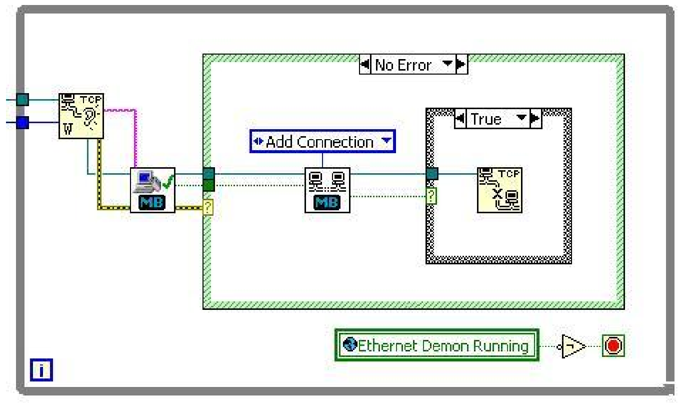


Figure 21: MODBUS TCP Slave Main Loop Part 1

The MODBUS TCP also needs to monitor the masters requests and respond to these requests to establish a connection; similarly to the serial slave.

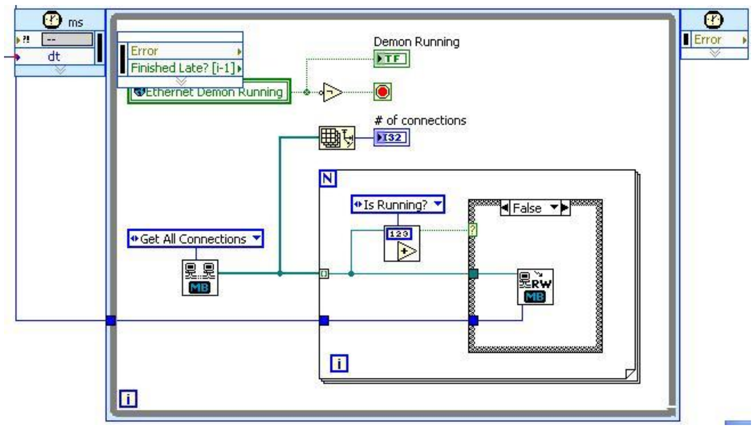


Figure 22: MODBUS TCP Slave Main Loop Part 2

MODBUS IO server

Please note that you can also turn your RT target into a MODBUS Slave using IO servers, part of the LabVIEW Datalogging and supervisory control module.

MODBUS Frequently Asked Questions

Q: Why is my MODBUS device timing out and not receiving responses from LabVIEW?
A: First, make sure that your serial port settings match the serial settings for the MODBUS device. Then check your device manual for the type of serial cable you need to communicate with the device. Some devices require straight-through cables while others require crossover cables.
Note: You must have NI-VISA installed to communicate with serial devices.
Tip: Make sure you select the serial port connected to your device when sending and receiving commands.

Q: How can I verify that my serial port is working correctly?
A: Refer to Serial Communication Starting Point for information and troubleshooting tips.

Q: The register values are not being updated on the MODBUS device or in LabVIEW, but the MODBUS device is not timing out. What could be the reason for this?
A: Make sure that you specify the correct address in the MODBUS device configuration software for the register you want to use in LabVIEW. In most MODBUS device configuration software, you must enter a name for the register you want to use. Per MODBUS convention, the register address of the slave device is calculated by subtracting 1 from the register name that you specify in the master device configuration software. The MODBUS LabVIEW library expects register addresses, not register names, so you may have to subtract 1 from the address you defined in the MODBUS device configuration software. For example, a register name defined as 2 in a MODBUS configuration device translates to register address 1 in the Holding Registers table of the LabVIEW MODBUS library.

MODBUS Device	Holding Register Name = 2
LabVIEW	Holding Register Address = 1

The MODBUS data model is based on a series of four tables: Discrete Inputs, Coils, Input Registers, and Holding Registers. These tables do not overlap in LabVIEW. Some MODBUS devices use the following start addresses for these tables.

0x00000 for the Coils
0x10000 for the Discrete Inputs
0x30000 for the Input Registers
0x40000 for the Holding Registers

Because the tables do not overlap in LabVIEW, ignore the first digit of the start addresses when defining the addresses in LabVIEW. For example, a register name defined as 0x40000 in a MODBUS configuration device translates to register address 0 in the LabVIEW Holding Registers table.

MODBUS Device	Holding Register Name = 0x40000
LabVIEW	Holding Register Address = 0

Sometimes you need to subtract 1 from the register name that you specify in the master device configuration software and ignore the first digit of the start address to ensure proper register addressing. For example, a register name defined as 0x40008 in a MODBUS configuration device translates to register address 7 in the LabVIEW Holding Registers table.

MODBUS Device	Holding Register Name = 0x40008
LabVIEW	Holding Register Address = 7

[1]Resources

Please note that the main resource used for this document is the MODBUS located at www.modbus.org.