IZMIR INSTITUTE OF TECHNOLOGY
IZTECH

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EE491 / EE492 PROJECT REPORT

MODBUS TCP-IP COMMUNICATION

İlker KESER -270206059

Yahya EKİN -260206053

ADVISOR: Ergün GÖZEK

DATE: 12/06/2024

ABSTRACT

Importance of communication protocols increases day by day due to expansion of machinery and automation in the factories. The adaptation of control systems to new machines and control interfaces requires a common language between all the controllers. This common language is MODBUS protocol. MODBUS is the most widespread communication protocol between various devices such as programmable logic controllers (PLC), human machine interface (HMI) panels, etc. PLC’s and HMI’s work together to control other elements and machinery in the factory. Human operator needs HMI panels to control the connected machinery by using controlling devices such as PLC’s. In this project, the descendent of the MODBUS protocol, MODBUS TCP/IP communication protocol implemented to generate complete communication between both master device and slave device. To represent the master device, a graphical user interface is designed in python programming language. To simulate the slave device, atmega368 based microcontroller and STM32 based microcontroller used. In both devices, the first 6 function of MODBUS TCP/IP protocol is implemented without using any external libraries. These functions are: Read Coils, Read Discrete Inputs, Read Holding Registers, Read Input Registers, Write Single Coil and Write Single Register. After the implementation, both devices are tested, and the communication is verified. Master program is tested with Schneider Electric PLC and the communication is established.

TABLE OF CONTENTS

ABSTRACT.. 1
TABLE OF CONTENTS.. 2
ABBREVIATIONS.. 3
LIST OF FIGURES.. 4
LIST OF TABLES.. 6
INTRODUCTION... 7
What is PLC?.. 7
Why need for MODBUS Protocol?.. 7
Development of MODBUS Protocol... 8
Example MODBUS Communication Structure.. 9
MODBUS Protocol Structure...10
PROBLEM DEFINITON...13
Problem Statement...13
PROPOSED SOLUTION...13
Master Device..14
Slave Application with Arduino NANO Atmega328P...15
....15
Function 1 Read Coils Arduino Code...16
Function 2 Read Discrete Inputs Arduino Code..18
Function 3 Read Holding Registers Arduino Code...21
Function 4 Read Input Registers Arduino Code..23
Function 5 Write Single Coil Arduino Code..25
Function 6 Write Single Register Arduino Code..27
ENGINEERING STANDARDS and DESIGN CONSTRAINTS......................................30
RESULTS AND DISCUSSIONS..31
Python Master – Schneider M221 PLC Slave..31
Python Master – Arduino Nano Slave..32
Python Master – STM32F407 Discovery Board Slave...32
CONCLUSIONS...34
REFERENCES...35

ABBREVIATIONS

ADU: Application Data Unit

HMI: Human Machine Interface

I/O: Input/Output

IP: Internet Protocol

MAC: Media Access Control

MB: MODBUS Protocol

MBAP: MODBUS Application Protocol

PDU: Protocol Data Unit PLC Programmable Logic Controller

TCP: Transmission Control Protocol

LIST OF FIGURES

Figure 1. M221 PLC (on left), example PLC wiring with various devices (on right). 7
Figure 2. Example MODBUS Connections. 9
Figure 3. Possible MODBUS Communications Scheme. 9
Figure 4. MODBUS TCP/IP Frame. 10
Figure 5. Example MODBUS interaction. 10
Figure 6. MODBUS Protocol Specifications. 11
Figure 7. Example MODBUS Function Structure. 11
Figure 8. Example MODBUS Function Flowchart. 12
***Table 1**. MODBUS functions and their purposes*..13
Figure 9. Master Program GUI. 14
Figure 10. Master Program Flowchart. 14
Figure 11: Arduino Nano Slave Device 15
Figure 12: Back Side of W5500..15
Figure *13*: Front side of W5500[]..15
Figure 14: General Algorithm 16
Figure 15: Request to read discrete outputs 20–38...16
Figure 16: Client (master) function 1 message output 17
Figure 17: Function 1's PDU Arduino code..18
Figure 18: Function 1's PDU Arduino code..18
Figure 19: Example of a request to read discrete inputs 197 – 218...18
Figure 20: Function 2 client message view..19
Figure 21. Function 2's Arduino Code. 20
Figure 11: Arduino Response message PDU code 20
Figure 22: Arduino Serial port output...20
Figure 23: Example of a request to read registers 108 – 110 21
Figure 24: Function 3 client message view..21
Figure 25: Function 3 Response PDU Arduino code 22
Figure 26: Function 3 Arduino Serial port output view 22
Figure 27: Example of a request to read input register 0-1..23
Figure 28: Function 4 client view...23
Figure 29: Function 4 PDU Arduino Code 24
Figure 30: Function 4 Arduino Serial port output view 24
Figure 31: Example of a request to write Coil 173 ON...25
Figure 32: Function 5 client view..25
Figure 33: Turning led with function 5's 0xFF00 value..26
Figure 34. Function 5 Response PDU Arduino Code..26
Figure 35. Arduino Serial Port Output View..27
Figure 36: example of a request to write register 2 to 00 03 hex 27
Figure 37: Function 6 client view..28
Figure 38: Function 6 Response PDU Arduino Code 29
Figure 39: Arduino Serial Monitor Output view 29

Figure 40. MODBUS communication among Python Master and M221 PLC Slave 31
Figure 41. Python Master, Arduino Nano Slave. 32
Figure 42. Python Master, STM32F4 Discovery Slave. 32
Figure 43. Hercules Master, STM32F4 Discovery Slave. 33

LIST OF TABLES

Table 1 MODBUS functions and their purposes................**Hata! Yer işareti tanımlanmamış.**13

INTRODUCTION

In industrial automation and control systems, the Modbus TCP/IP protocol is a commonly used communication standard. It was first created in the late 1970s for use in programmable logic controllers (PLCs) by Modicon, which is now part of Schneider Electric. An extension of the Modbus protocol designed for TCP/IP networks; Modbus TCP/IP enables smooth device-to-device communication via Ethernet.

What is PLC?

PLCs, or programmable logic controllers, are industrial computers that are frequently employed in control and automation systems. They are intended to execute specialized functions and govern a variety of industrial operations. PLCs consist of CPU, input/output modules which might be transistor or relay based, and programming device. The origin of the PLCs starts in the late 1960s as a replacement for relay based electromechanical devices. Modicon 084 was the first PLC in the industry which is developed by Bedford Associates in 1968. After they gain extensive usage in industry, they are transformed from simple on-off devise to more powerful, more advanced and modular devices.

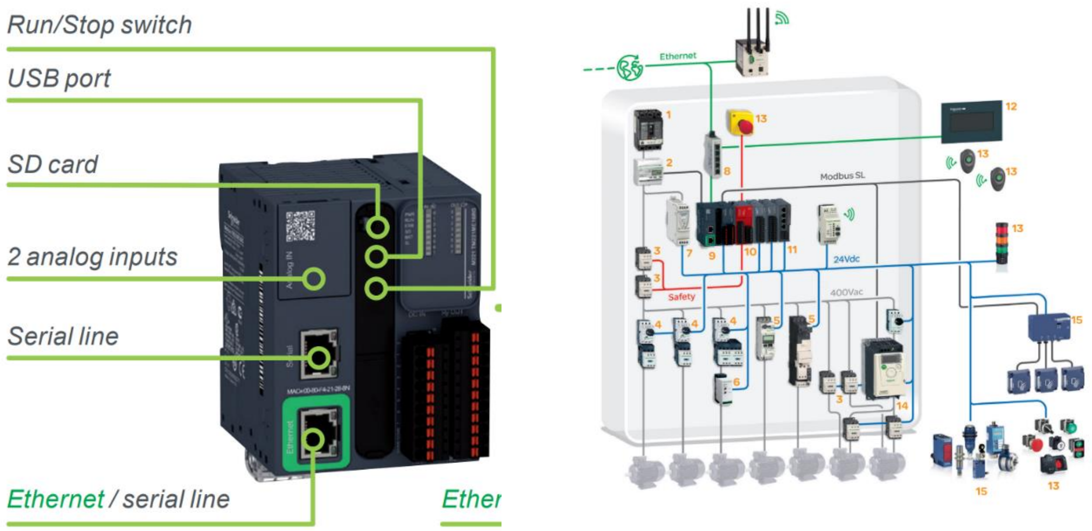


Figure 1. M221 PLC (on left), example PLC wiring with various devices (on right).

Why need for MODBUS Protocol?

After the development of various kind of PLCs, the need for communication between them occurred due to application reasons. As a simple example, a factory which had X brand PLC should had to buy the same brand to establish communication between them. Since there are numerous brands in the industry, to establish communication among different brands MODBUS protocol is developed. By this development, the PLC industry is grooved, and PLCs became more advanced than before.

Development of MODBUS Protocol

In 1979, Modicon (now Schneider Electric) developed the Modbus protocol specifically for use with their programmable logic controllers (PLCs). It was initially intended for serial communication using RS-232, RS-485, and RS-422 interfaces. Since then, Modbus has developed to include several additional communication protocols, including as Ethernet and TCP/IP. Modicon first released Modbus in 1979 for use with their PLCs, or programmable logic controllers. It was created to monitor and manage these PLCs using a serial link to a Human Machine Interface (HMI). A straightforward, vendor-neutral communication protocol served as the foundation for the original Modbus protocol, making it simple to integrate with a variety of systems and devices. Modbus TCP/IP is a modification of the Modbus protocol designed for usage in TCP/IP networks. Its purpose was to provide smooth connection and data sharing across different parts and devices in an industrial network. Based on Ethernet technology, Modbus TCP/IP enables many masters to connect to a single slave as long as separate TCP ports are used.

Key Features and Advantages of MODBUS TCP/IP

Because of its well-known emphasis on durability and simplicity, Modbus TCP/IP is a preferred option for many industrial applications. Important characteristics consist of:

Slave/Master Architecture: A master makes requests to a slave, which responds to the master and completes the action, in the master/slave communication model used by Modbus TCP/IP.

Message Structure: An application data unit (ADU) and a protocol data unit (PDU) make up a Modbus message. Whereas the PDU houses the function code and data, the ADU houses the address and error checking code.

Ethernet Support: Since Ethernet hardware is the foundation of Modbus TCP/IP, multiple masters can connect to the same slave as long as separate TCP ports are used.

Example MODBUS Communication Structure

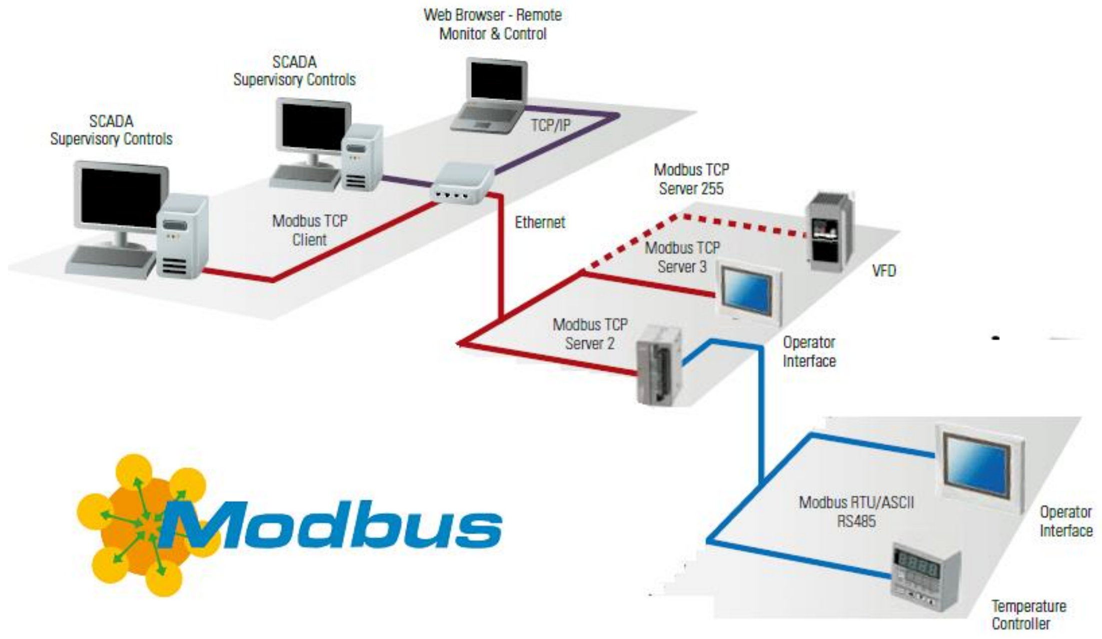


Figure 2. Example MODBUS Connections.

MODBUS communication allows connection between various devices such as computers, human-machine interface units, temperature controllers, servers as shown in figure above. Moreover, the electric motors valves, pressure tanks, various kind of sensors can be connected together and controlled through MODBUS protocol. The possible connection and communication architecture is shown below in figure X.

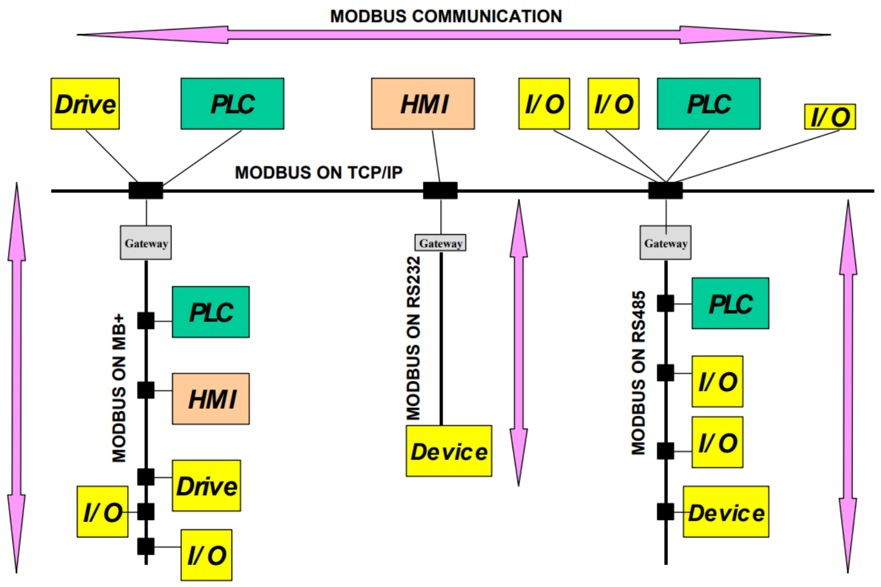


Figure 3. Possible MODBUS Communications Scheme.

MODBUS Protocol Structure

MODBUS protocol uses Master-Slave configuration. The communication process starts by request of the master device. This request is indicated in slave device and response is generated and send back to the master. Then, the incoming response is confirmed from master. In MODBUS TCP/IP communication, data is sent in protocol data unit, which consists of function code and transmitted data in the order which is specified by MODBUS protocol. Then the MBAP header is added to PDU. The concatenation of MBAP header and PDU generates the MODBUS TCP frame. The resulting frame is send via using TCP/IP protocol. The frame structure is showed below in figure X.

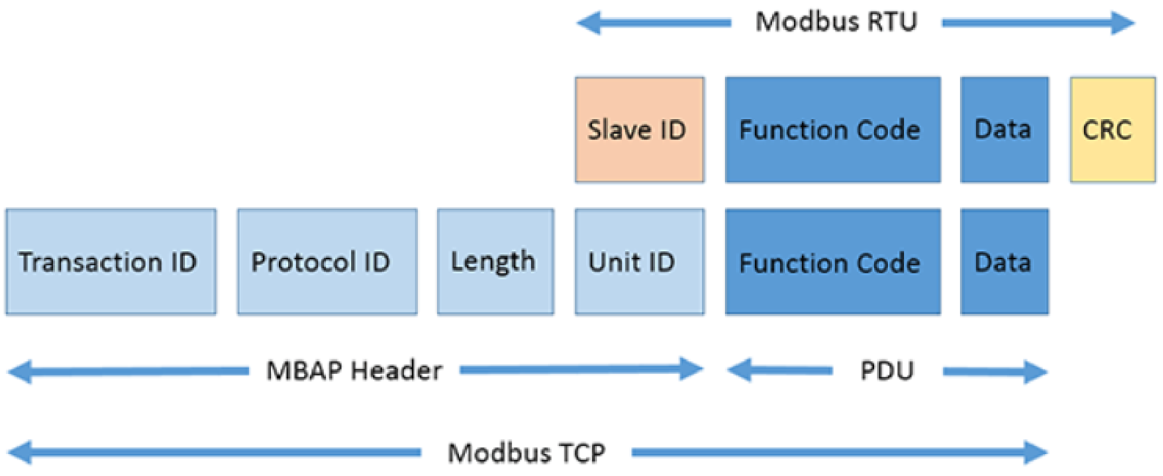


Figure 4. MODBUS TCP/IP Frame.

In all requests and responses between master (client in figure 5) and slave device (server in figure 5) same frame structure is used. The example interaction between master and slave shown in figure 5.

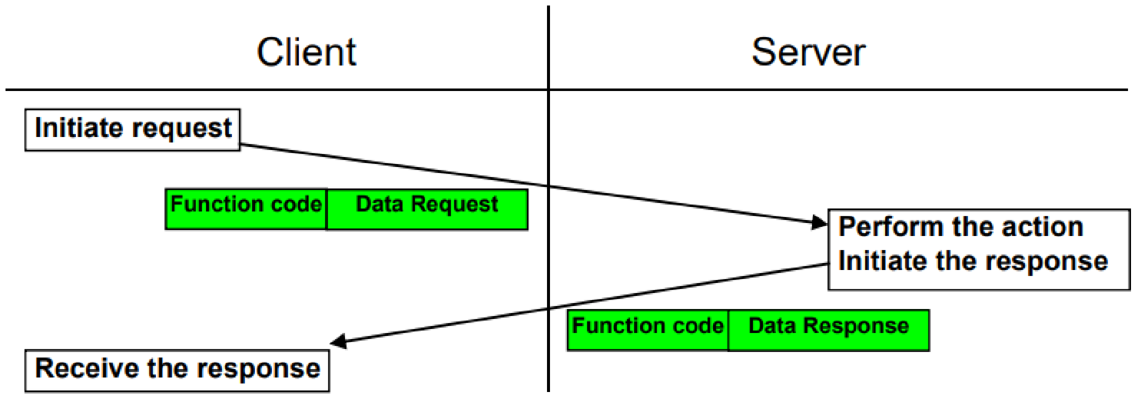
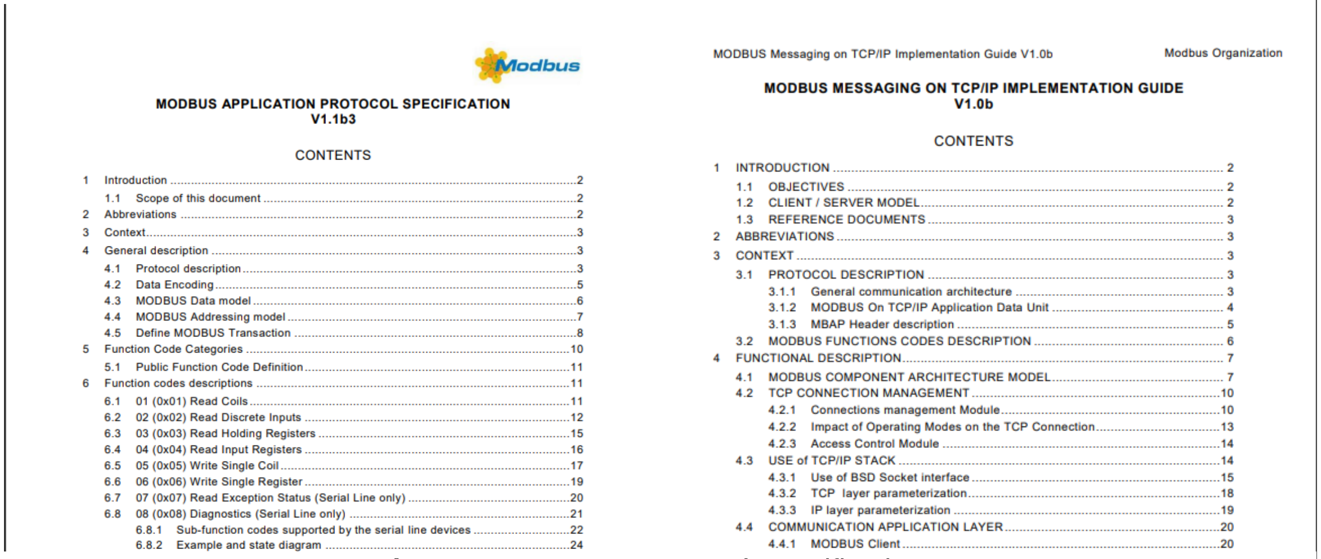


Figure 5. Example MODBUS interaction.

This shows the complete MODBUS communication. The request and response data is defined in MODBUS Application Protocol Specification V1.1b3.

The implementation of MODBUS protocol and TCP/IP messaging implementation is described in those two documents from MODBUS organization.



MODBUS APPLICATION PROTOCOL SPECIFICATION
V1.1b3

CONTENTS

1 Introduction 2
1.1 Scope of this document 2
2 Abbreviations 2
3 Context 3
4 General description 3
4.1 Protocol description 3
4.2 Data Encoding 5
4.3 MODBUS Data model 6
4.4 MODBUS Addressing model 7
4.5 Define MODBUS Transaction 8
5 Function Code Categories 10
5.1 Public Function Code Definition 11
6 Function codes descriptions 11
6.1 01 (0x01) Read Coils 11
6.2 02 (0x02) Read Discrete Inputs 12
6.3 03 (0x03) Read Holding Registers 15
6.4 04 (0x04) Read Input Registers 16
6.5 05 (0x05) Write Single Coil 17
6.6 06 (0x06) Write Single Register 19
6.7 07 (0x07) Read Exception Status (Serial Line only) 20
6.8 08 (0x08) Diagnostics (Serial Line only) 21
6.8.1 Sub-function codes supported by the serial line devices 22
6.8.2 Example and state diagram 24

MODBUS Messaging on TCP/IP Implementation Guide V1.0b   Modbus Organization

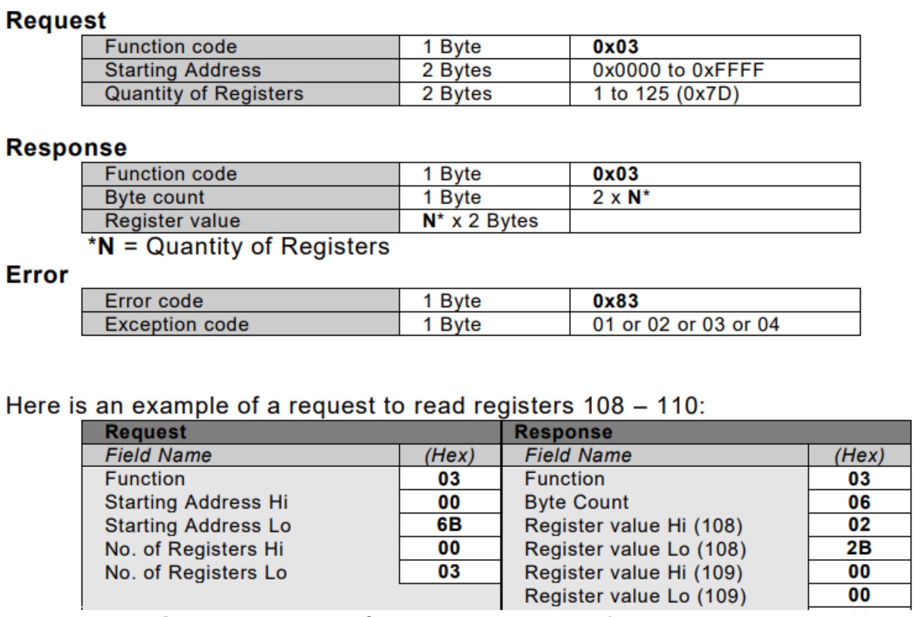
MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE
V1.0b

CONTENTS

1 INTRODUCTION 2
1.1 OBJECTIVES 2
1.2 CLIENT / SERVER MODEL 2
1.3 REFERENCE DOCUMENTS 3
2 ABBREVIATIONS 3
3 CONTEXT 3
3.1 PROTOCOL DESCRIPTION 3
3.1.1 General communication architecture 3
3.1.2 MODBUS On TCP/IP Application Data Unit 4
3.1.3 MBAP Header description 5
3.2 MODBUS FUNCTIONS CODES DESCRIPTION 6
4 FUNCTIONAL DESCRIPTION 7
4.1 MODBUS COMPONENT ARCHITECTURE MODEL 7
4.2 TCP CONNECTION MANAGEMENT 10
4.2.1 Connections management Module 10
4.2.2 Impact of Operating Modes on the TCP Connection 13
4.2.3 Access Control Module 14
4.3 USE of TCP/IP STACK 14
4.3.1 Use of BSD Socket interface 15
4.3.2 TCP layer parameterization 18
4.3.3 IP layer parameterization 19
4.4 COMMUNICATION APPLICATION LAYER 20
4.4.1 MODBUS Client 20

Figure 6. MODBUS Protocol Specifications.

In the documents above, the structure of the functions are given by flowcharts and example request and response messages. To implement the protocol, both master and slave device code should be written according to these definitions and flowcharts. The one of the example



Request

Function code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x **N***
Register value	**N*** x 2 Bytes	

***N** = Quantity of Registers

Error

Error code	1 Byte	**0x83**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read registers 108 – 110:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**03**	Function	**03**
Starting Address Hi	**00**	Byte Count	**06**
Starting Address Lo	**6B**	Register value Hi (108)	**02**
No. of Registers Hi	**00**	Register value Lo (108)	**2B**
No. of Registers Lo	**03**	Register value Hi (109)	**00**
		Register value Lo (109)	**00**

Figure 7. Example MODBUS Function Structure.

definitions, Read Holding Registers function is described in the protocol specification document as below:

The flowchart of the communication is also given in document, which is represented below.

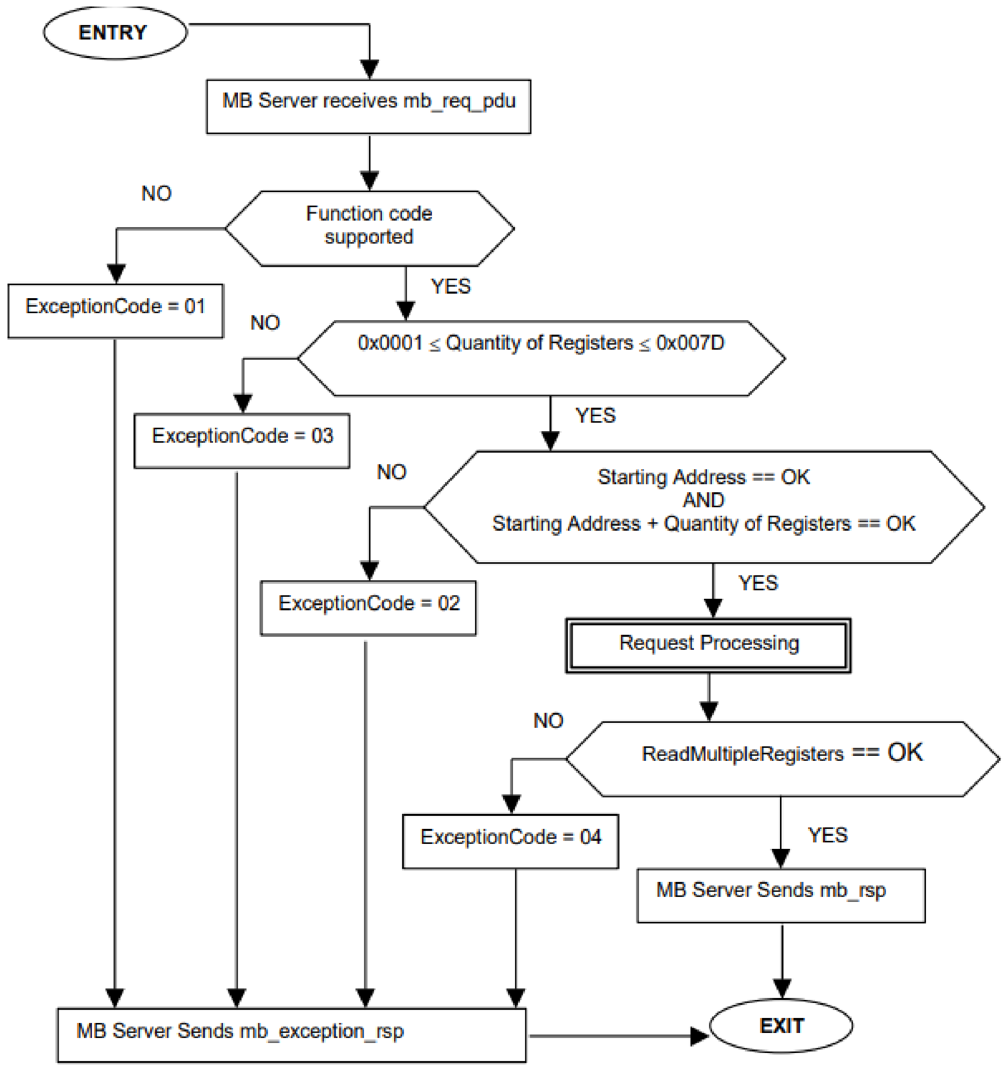

ENTRY
MB Server receives mb_req_pdu
Function code supported
NO
YES
ExceptionCode = 01
0x0001 ≤ Quantity of Registers ≤ 0x007D
NO
YES
ExceptionCode = 03
Starting Address == OK AND Starting Address + Quantity of Registers == OK
NO
YES
ExceptionCode = 02
Request Processing
ReadMultipleRegisters == OK
NO
YES
ExceptionCode = 04
MB Server Sends mb_rsp
MB Server Sends mb_exception_rsp
EXIT

Figure 8. Example MODBUS Function Flowchart.

PROBLEM DEFINITON

Problem Statement

In the world of industrial automation, where precision and efficiency are critical, there's a growing demand for advanced monitoring and control systems. These systems must not only monitor parameters in real-time but also control analog and digital inputs, and generate digital and analog outputs. They need to ensure seamless communication and adaptability to the fast-paced industrial environment. As an example factory, the need of reading sensor data and controlling the input/output of the production devices from the computer using graphical user interface is occurred. The expected communication system consists of 1 master device which has graphical user interface to perform the first 6 function of MODBUS TCP/IP protocol and 1 slave device which has capability of performing the first 6 function of MODBUS TCP/IP protocol needs to be designed. The design constraint is using MODBUS libraries from anywhere for both system is forbidden.

The first 6 function of MODBUS protocol and their purpose is given in table X below.

***Table 1**. MODBUS functions and their purposes*.

Function	Purpose
Read Coils	Reads the status of connected coils in remote device.
Read Discrete Inputs	Reads the status of connected digital inputs in remote device.
Red Holding Registers	Reads the memory (holding) registers of remote device.
Read Input Registers	Reads the input registers (analog inputs) of the remote device.
Write Single Coil	Drives the connected coil on remote device.
Write Single Register	Writes data to memory(holding) registers of remote device.

PROPOSED SOLUTION

The solution consists of developing master device on python language. To generate the user interface, tkinter module of the python is used. The GUI performed the first 6 functions of MODBUS protocol. To develop a slave device, Arduino Nano is selected. Due to simulation of the connected devices to slave device, 2 LED is used to represent 2 connected coil. 2 push-button used to simulate 2 discrete inputs. Also, 2 potentiometer is used as analog inputs to slave device.

Master Device

Master program’s graphical user interface is explained below. This section does not include brief description of the functions since these details are same with the slave code. In the slave description part, both the graphical user interface usage and slave response is presented.

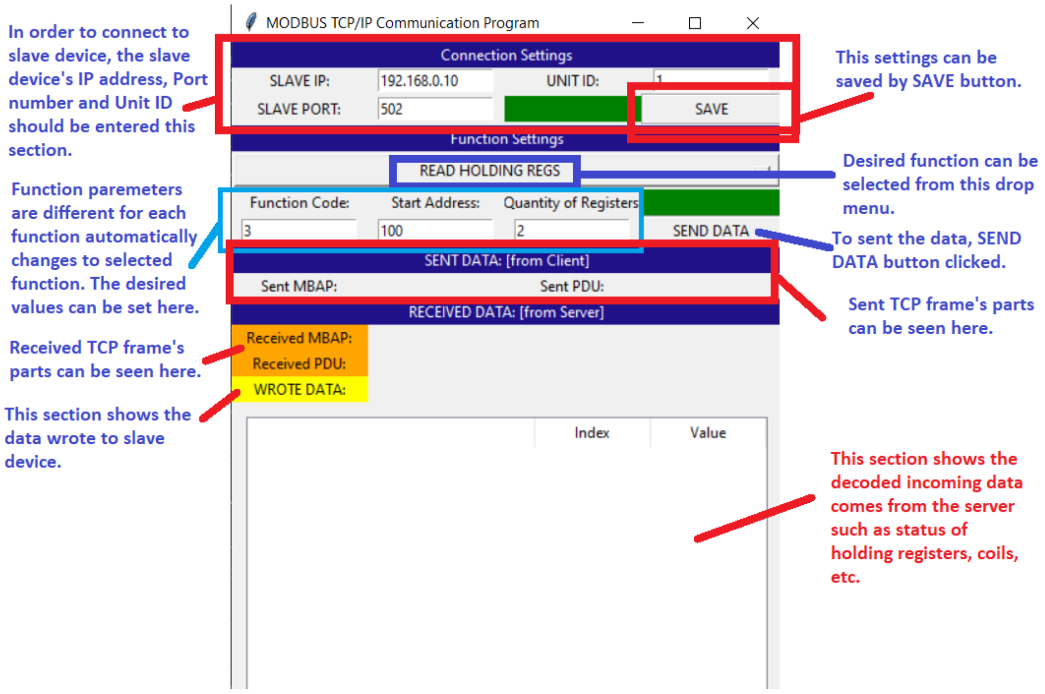

In order to connect to slave device, the slave device's IP address, Port number and Unit ID should be entered this section.
Function paremeters are different for each function automatically changes to selected function. The desired values can be set here.
Received TCP frame's parts can be seen here.
This section shows the data wrote to slave device.
MODBUS TCP/IP Communication Program
Connection Settings
SLAVE IP:
192.168.0.10
UNIT ID:
1
SLAVE PORT:
502
SAVE
Function Settings
READ HOLDING REGS
Function Code:
Start Address:
Quantity of Registers
3
100
2
SEND DATA
SENT DATA: [from Client]
Sent MBAP:
Sent PDU:
RECEIVED DATA: [from Server]
Received MBAP:
Received PDU:
WROTE DATA:
Index
Value
This settings can be saved by SAVE button.
Desired function can be selected from this drop menu.
To sent the data, SEND DATA button clicked.
Sent TCP frame's parts can be seen here.
This section shows the decoded incoming data comes from the server such as status of holding registers, coils, etc.

Figure 9. Master Program GUI.

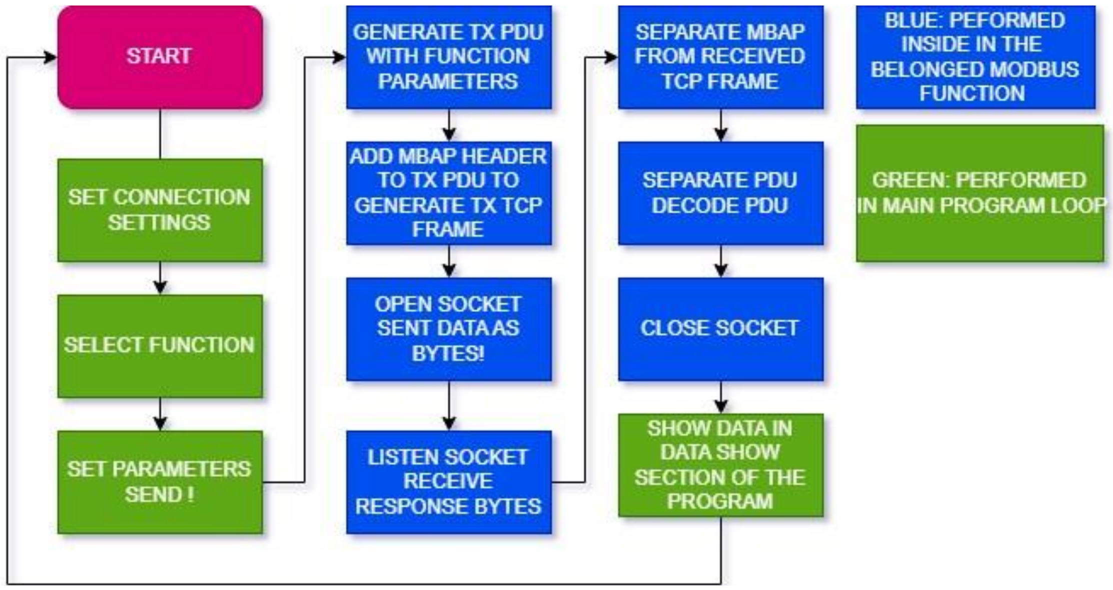

START
SET CONNECTION SETTINGS
SELECT FUNCTION
SET PARAMETERS SEND !
GENERATE TX PDU WITH FUNCTION PARAMETERS
ADD MBAP HEADER TO TX PDU TO GENERATE TX TCP FRAME
OPEN SOCKET SENT DATA AS BYTES!
LISTEN SOCKET RECEIVE RESPONSE BYTES
SEPARATE MBAP FROM RECEIVED TCP FRAME
SEPARATE PDU DECODE PDU
CLOSE SOCKET
SHOW DATA IN DATA SHOW SECTION OF THE PROGRAM
BLUE: PEFORMED INSIDE IN THE BELONGED MODBUS FUNCTION
GREEN: PERFORMED IN MAIN PROGRAM LOOP

Figure 10. Master Program Flowchart.

Slave Application with Arduino NANO Atmega328P

We designed a slave device with Atmega328P based Arduino Nano. In order for this slave device to communicate with the MODBUS protocol, it needs to make an extra ethernet communication, because we send the data over the ethernet cable. Arduino Nano doesn't have an ethernet capability. We added a WIZnet W5500 ethernet module. The reason we choose is its library is very easy to use and implement to microcontroller.

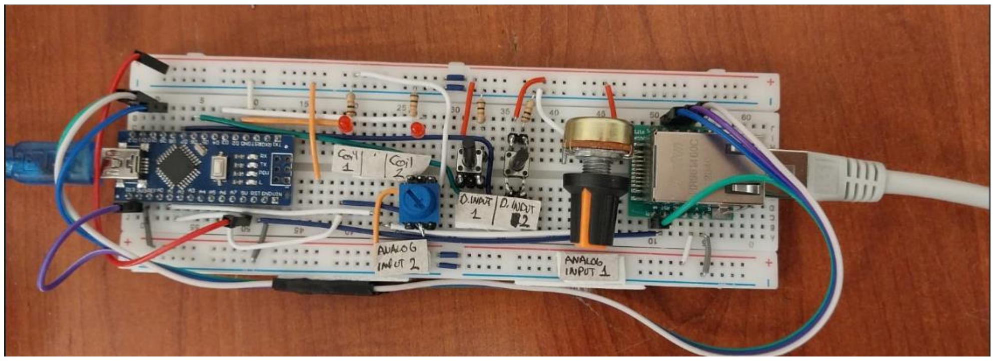


Figure 11: Arduino Nano Slave Device

Figure 12: Back Side of W5500

Figure *13*: Front side of W5500[]

In Figure 11, we made a test slave device on breadboard. Arduino connects to computer with ethernet cable. There are some buttons, potentiometers and LEDs. These components are used to test for code's output. The device shown in Figure 12 is back side of W5500 ethernet module. There is a W5500 ethernet controllers. Figure 13 is view of front side of W5500. In module it uses Hanrun RJ45 connector.

When we look at the codes of the Arduino, we used producer's libraries. Socket and W5500 libraries are worked compatible. MODBUS's functions we write are implemented easily.

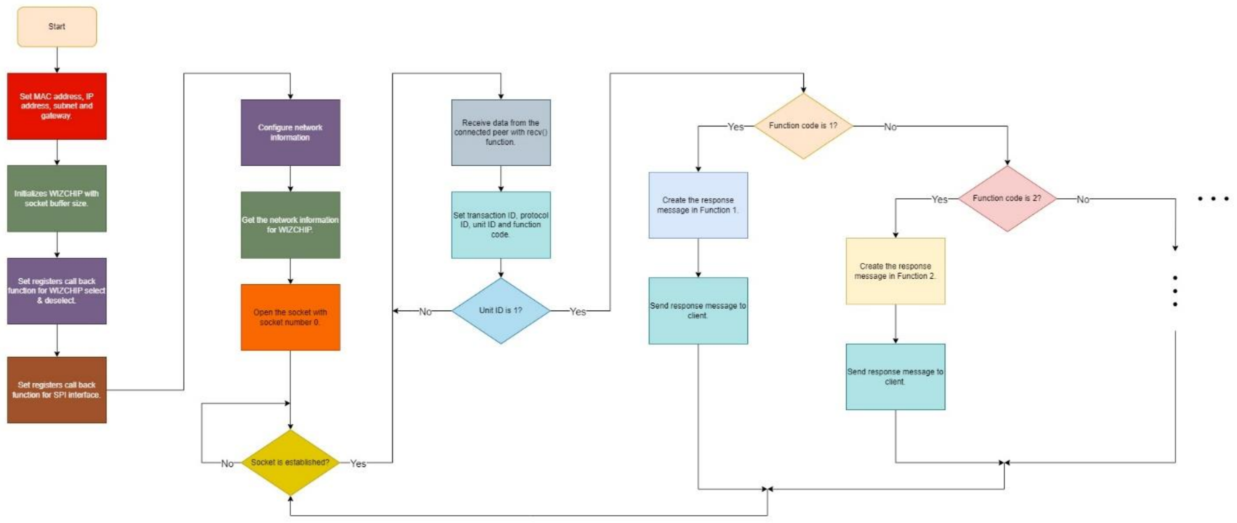
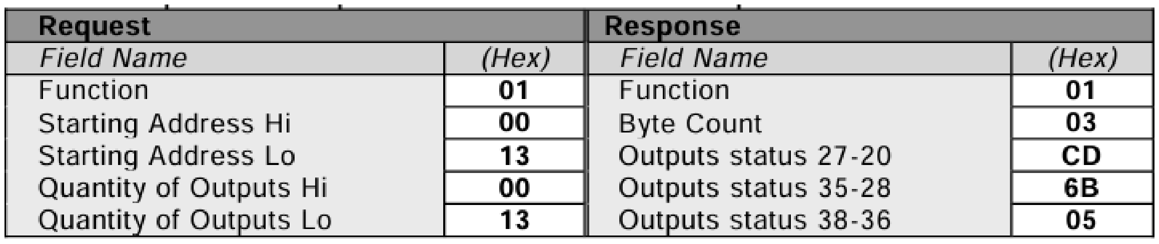

Start
Set MAC address, IP address, subnet and gateway.
Initializes WIZCHIP with socket buffer size.
Set registers call back function for WIZCHIP select & deselect.
Set registers call back function for SPI interface.
Configure network information
Get the network information for WIZCHIP.
Open the socket with socket number 0.
Socket is established?
No
Yes
Receive data from the connected peer with recv() function.
Set transaction ID, protocol ID, unit ID and function code.
Unit ID is 1?
No
Yes
Function code is 1?
Yes
No
Create the response message in Function 1.
Send response message to client.
Function code is 2?
Yes
No
Create the response message in Function 2.
Send response message to client.

Figure 14: General Algorithm

In Figure 14, software implementation of MODBUS TCP-IP communication to Arduino Nano. First 6 functions MODBUS are implemented successfully.

Function 1 Read Coils Arduino Code

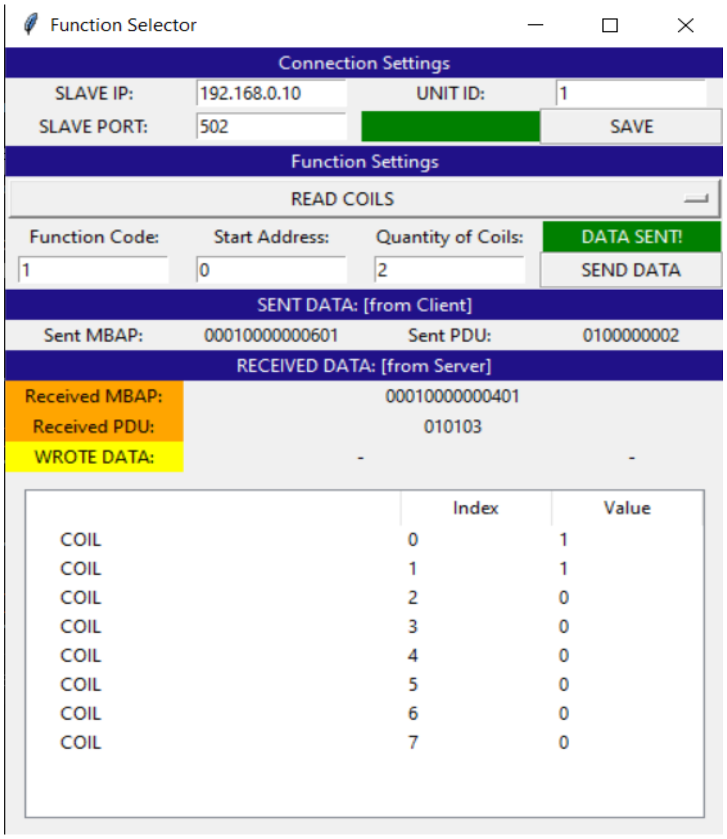
This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. The PDU request specifies the start address of the first DO register and the subsequent number of required DO values. In the PDU, the DO values are addressed starting from zero. Example PDU output of function 1 is shown below (Figure 5):



Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**01**	Function	**01**
Starting Address Hi	**00**	Byte Count	**03**
Starting Address Lo	**13**	Outputs status 27-20	**CD**
Quantity of Outputs Hi	**00**	Outputs status 35-28	**6B**
Quantity of Outputs Lo	**13**	Outputs status 38-36	**05**

Figure 15: Request to read discrete outputs 20–38

In Figure 15, the PDU part of the request message starts with the function code, continues with the high-low part of the start address, and ends with the high-low part of the output number. This is a one-byte message in total message’s every part, including all reserved parts. Request message came from master (client) device. As you saw the below, this is our master message GUI (Graphical User Interface) program screen (Figure 16).



SLAVE IP:	192.168.0.10	UNIT ID:	1
SLAVE PORT:	502	🟩	SAVE

Function Code:	Start Address:	Quantity of Coils:	DATA SENT!
1	0	2	SEND DATA

Sent MBAP:	00010000000601	Sent PDU:	0100000002

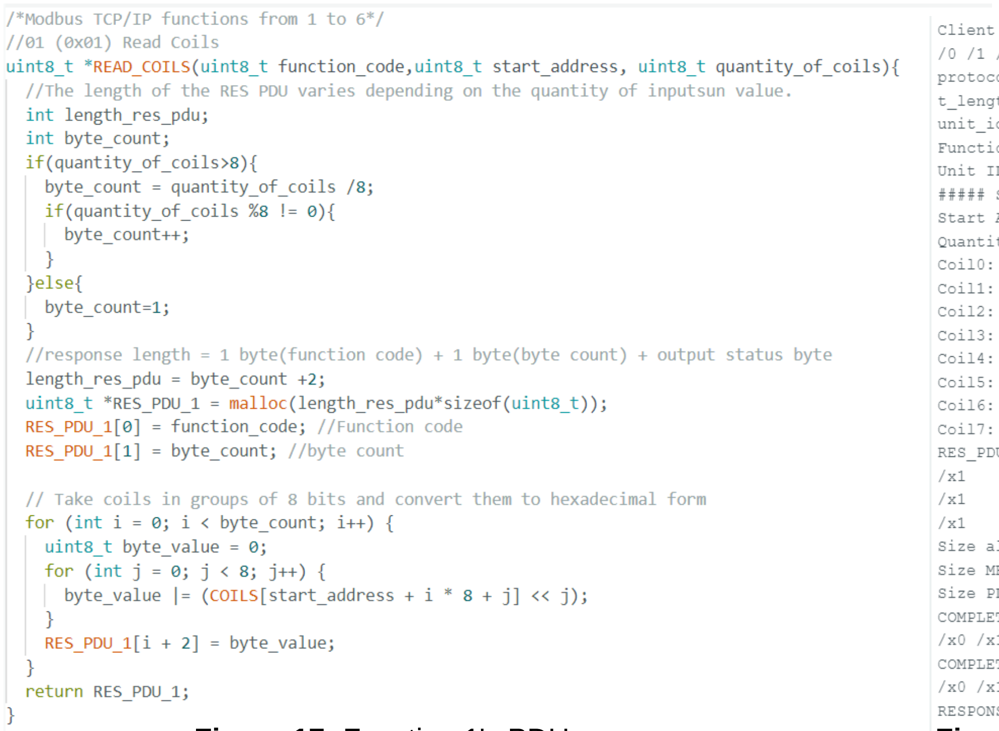
Received MBAP:	00010000000401	
Received PDU:	010103	
WROTE DATA:	-	-

	Index	Value
COIL	0	1
COIL	1	1
COIL	2	0
COIL	3	0
COIL	4	0
COIL	5	0
COIL	6	0
COIL	7	0

Figure 16: Client (master) function 1 message output

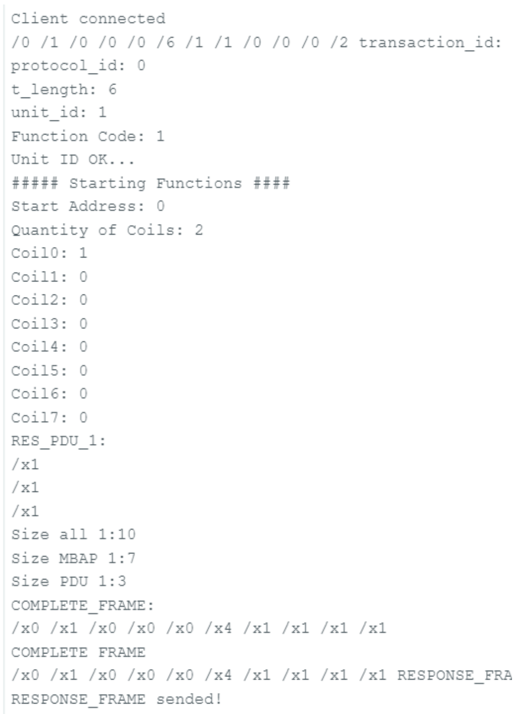
When we write function code, start address and quantity of coils, MBAP and PDU message part can be occurs automatically and send to server.

In Figure 17, PDU message part of slave is shown below. Function code, start address and quantity of coils come from request message. First, we determine the byte count. If the input values are greater than 8, the number of bytes is found by dividing by 8. If a remainder occurs, another byte is created and add to created sum of byte number. Dynamic memory management is performed according to the number of bytes created. As we know, response message part's first two byte (Figure 15) are function code and byte count. Other bytes are converted to hexadecimal of status of coils' high/low value.



```
/*Modbus TCP/IP functions from 1 to 6*/
//01 (0x01) Read Coils
uint8_t *READ_COILS(uint8_t function_code,uint8_t start_address, uint8_t quantity_of_coils){
  //The length of the RES PDU varies depending on the quantity of inputsun value.
  int length_res_pdu;
  int byte_count;
  if(quantity_of_coils>8){
    byte_count = quantity_of_coils /8;
    if(quantity_of_coils %8 != 0){
      byte_count++;
    }
  }else{
    byte_count=1;
  }
  //response length = 1 byte(function code) + 1 byte(byte count) + output status byte
  length_res_pdu = byte_count +2;
  uint8_t *RES_PDU_1 = malloc(length_res_pdu*sizeof(uint8_t));
  RES_PDU_1[0] = function_code; //Function code
  RES_PDU_1[1] = byte_count; //byte count

  // Take coils in groups of 8 bits and convert them to hexadecimal form
  for (int i = 0; i < byte_count; i++) {
    uint8_t byte_value = 0;
    for (int j = 0; j < 8; j++) {
      byte_value |= (COILS[start_address + i * 8 + j] << j);
    }
    RES_PDU_1[i + 2] = byte_value;
  }
  return RES_PDU_1;
}
```

Figure 17: Function 1's PDU Arduino code



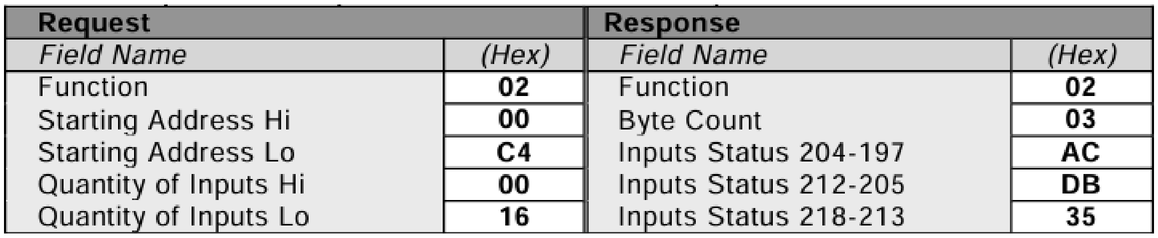
```
Client connected
/0 /1 /0 /0 /0 /6 /1 /1 /0 /0 /0 /2 transaction_id:
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 1
Unit ID OK...
##### Starting Functions ####
Start Address: 0
Quantity of Coils: 2
Coil0: 1
Coil1: 0
Coil2: 0
Coil3: 0
Coil4: 0
Coil5: 0
Coil6: 0
Coil7: 0
RES_PDU_1:
/x1
/x1
/x1
Size all 1:10
Size MBAP 1:7
Size PDU 1:3
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x4 /x1 /x1 /x1 /x1
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x4 /x1 /x1 /x1 /x1 RESPONSE_FRA
RESPONSE_FRAME sended!
```

Figure 18: Function 1's PDU Arduino code

When we send request function 1 message from client (master) to server (slave) the output of Arduino serial port is shown below (Figure 18).

Function 2 Read Discrete Inputs Arduino Code

This function code is utilized to retrieve the status of 1 to 2000 consecutive coils in a slave device. The Request PDU delineates the initial address, representing the address of the first input specified, along with the total number of inputs. The PDU of the response message specifies the function code, byte count, and status of discrete inputs (Figure 19).

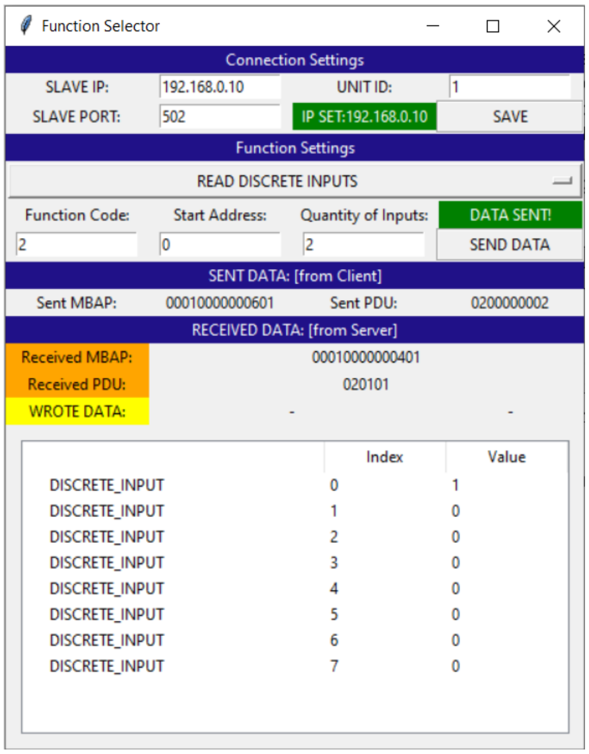


Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**02**	Function	**02**
Starting Address Hi	**00**	Byte Count	**03**
Starting Address Lo	**C4**	Inputs Status 204-197	**AC**
Quantity of Inputs Hi	**00**	Inputs Status 212-205	**DB**
Quantity of Inputs Lo	**16**	Inputs Status 218-213	**35**

Figure 19: Example of a request to read discrete inputs 197 – 218

In Figure 19, the PDU part of the request message starts with the function code, continues with the high-low part of the start address, and ends with the high-low part of the output number.

The status of discrete inputs 204–197 is represented by the byte value AC in hexadecimal, or 1010 1100 in binary. In this byte, input 204 is the most significant bit and input 197 is the least significant bit. The status of discrete inputs 218–213 is represented by the byte value 35 in hexadecimal, or 0011 0101 in binary. In this byte, input 218 is in the third bit position from the left, and input 213 is the LSB. The client's message is shown in Figure 20.


Function Selector

Connection Settings

SLAVE IP:	192.168.0.10	UNIT ID:	1
SLAVE PORT:	502	IP SET:192.168.0.10	SAVE

Function Settings

READ DISCRETE INPUTS

Function Code:	Start Address:	Quantity of Inputs:	DATA SENT!
2	0	2	SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 0200000002

RECEIVED DATA: [from Server]

Received MBAP: 00010000000401
Received PDU: 020101
WROTE DATA: - -

	Index	Value
DISCRETE_INPUT	0	1
DISCRETE_INPUT	1	0
DISCRETE_INPUT	2	0
DISCRETE_INPUT	3	0
DISCRETE_INPUT	4	0
DISCRETE_INPUT	5	0
DISCRETE_INPUT	6	0
DISCRETE_INPUT	7	0

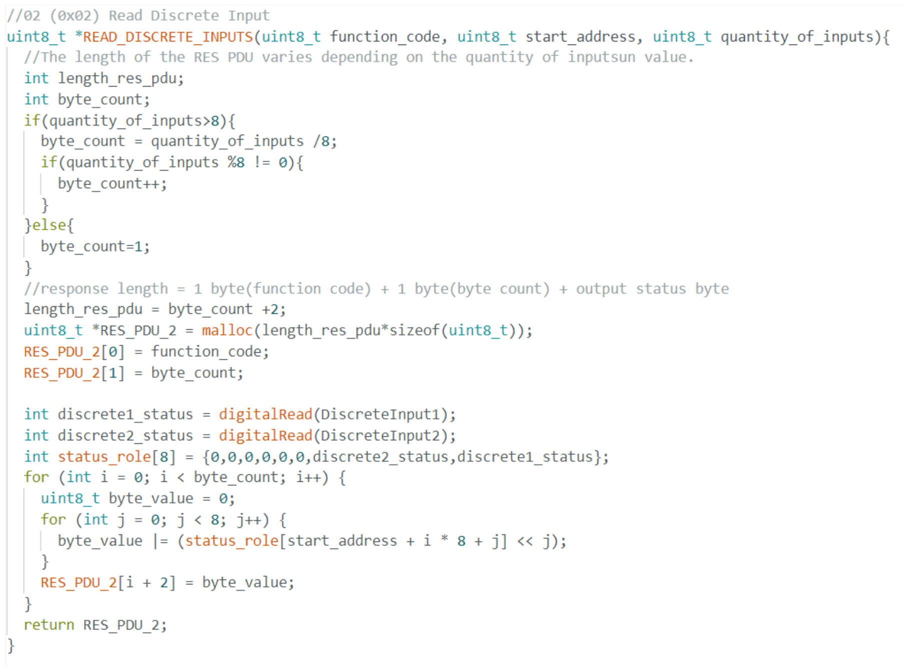
Figure 20: Function 2 client message view

When we push to buttons (Figure 11), the discrete input 1 is set high. If we push both buttons first two bits are set high. We can assign the starting address ourselves and we can find the status of discrete inputs.

When we write function code, start address and quantity of coils, MBAP and PDU message part can be occurs automatically and send to server.

In Figure 20, PDU message part of slave is shown below. Like function 1, we need to function code, start address and quantity of inputs from client's request message. After determine byte count, we set the first two response PDU's byte as function code and byte count.

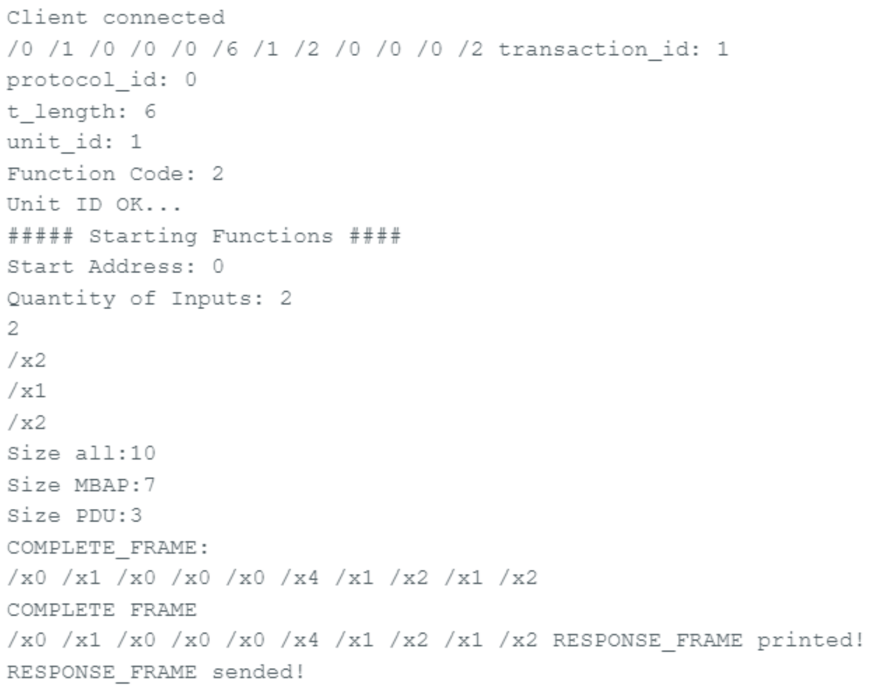
Since we are using two buttons, we learn the status of the two buttons from the *'digitalRead'* function. We pass the output of the two states into an array. Since the states will be high or low, we can convert the value of the array to a hexadecimal value. The reason we do this is because we cannot use discrete input enough.



```
//02 (0x02) Read Discrete Input
uint8_t *READ_DISCRETE_INPUTS(uint8_t function_code, uint8_t start_address, uint8_t quantity_of_inputs){
  //The length of the RES PDU varies depending on the quantity of inputsun value.
  int length_res_pdu;
  int byte_count;
  if(quantity_of_inputs>8){
    byte_count = quantity_of_inputs /8;
    if(quantity_of_inputs %8 != 0){
      byte_count++;
    }
  }else{
    byte_count=1;
  }
  //response length = 1 byte(function code) + 1 byte(byte count) + output status byte
  length_res_pdu = byte_count +2;
  uint8_t *RES_PDU_2 = malloc(length_res_pdu*sizeof(uint8_t));
  RES_PDU_2[0] = function_code;
  RES_PDU_2[1] = byte_count;

  int discrete1_status = digitalRead(DiscreteInput1);
  int discrete2_status = digitalRead(DiscreteInput2);
  int status_role[8] = {0,0,0,0,0,0,discrete2_status,discrete1_status};
  for (int i = 0; i < byte_count; i++) {
    uint8_t byte_value = 0;
    for (int j = 0; j < 8; j++) {
      byte_value |= (status_role[start_address + i * 8 + j] << j);
    }
    RES_PDU_2[i + 2] = byte_value;
  }
  return RES_PDU_2;
}
```

Figure 21. Function 2's Arduino Code.

When we send request function 2 message from client to server the output of Arduino is shown below (Figure 22).



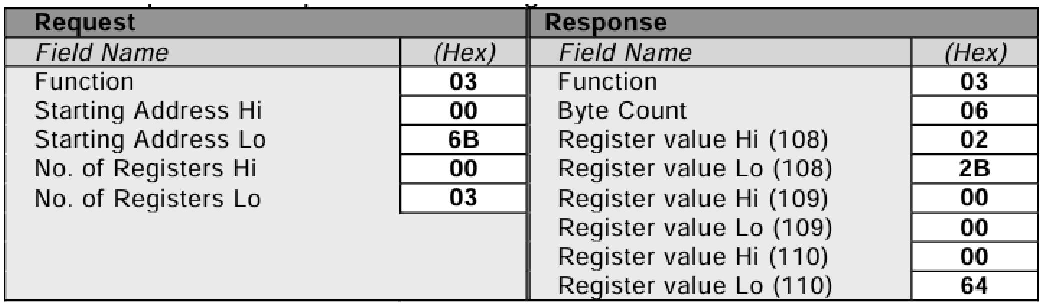
```
Client connected
/0 /1 /0 /0 /0 /6 /1 /2 /0 /0 /0 /2 transaction_id: 1
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 2
Unit ID OK...
##### Starting Functions ####
Start Address: 0
Quantity of Inputs: 2
2
/x2
/x1
/x2
Size all:10
Size MBAP:7
Size PDU:3
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x4 /x1 /x2 /x1 /x2
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x4 /x1 /x2 /x1 /x2 RESPONSE_FRAME printed!
RESPONSE_FRAME sended!
```

Figure 22: Arduino Serial port output

Function 3 Read Holding Registers Arduino Code

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers (Figure 23).

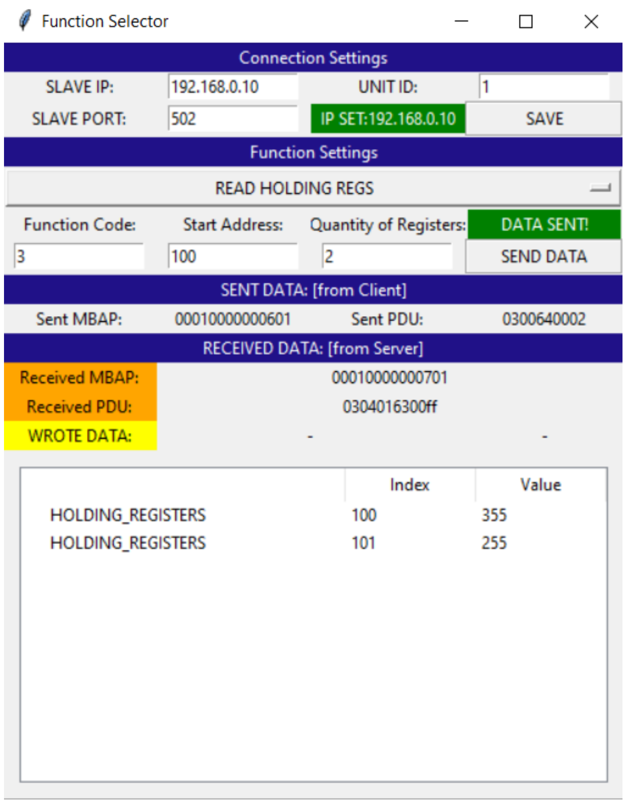
The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the most significant bits, and the second byte contains the least significant bits.



Request		Response	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**03**	Function	**03**
Starting Address Hi	**00**	Byte Count	**06**
Starting Address Lo	**6B**	Register value Hi (108)	**02**
No. of Registers Hi	**00**	Register value Lo (108)	**2B**
No. of Registers Lo	**03**	Register value Hi (109)	**00**
		Register value Lo (109)	**00**
		Register value Hi (110)	**00**
		Register value Lo (110)	**64**

Figure 23: Example of a request to read registers 108 – 110

The contents of register 108 are represented by the two-byte values 02 2B in hexadecimal, or 555 in decimal. The contents of registers 109 and 110 are 00 00 and 00 64 in hexadecimal, or 0 and 100 in decimal, respectively. The client message is shown in Figure 24.



SLAVE IP: 192.168.0.10 UNIT ID: 1
SLAVE PORT: 502 IP SET:192.168.0.10

READ HOLDING REGS

Function Code: 3
Start Address: 100
Quantity of Registers: 2
DATA SENT!

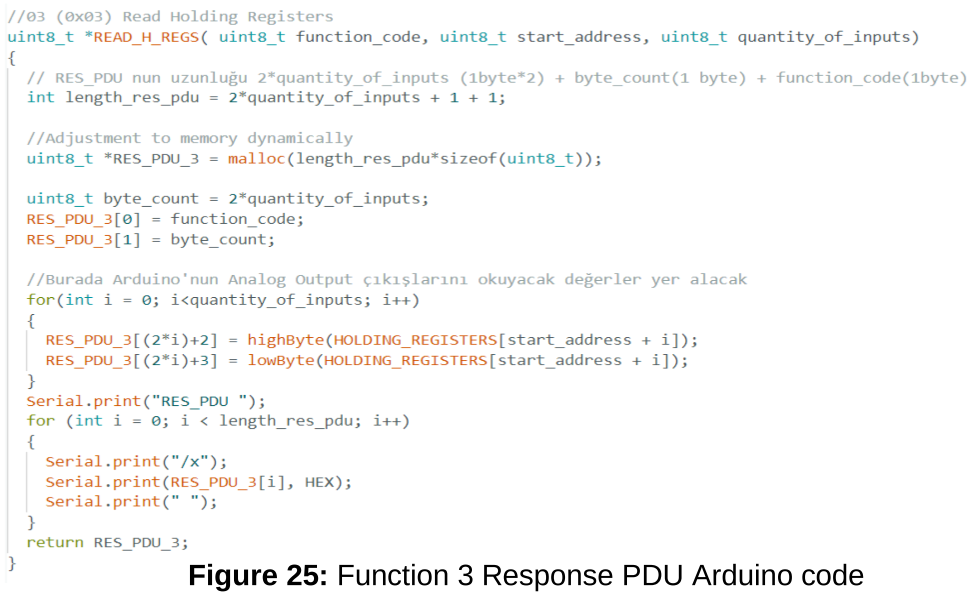
SENT DATA: [from Client]
Sent MBAP: 00010000000601 Sent PDU: 0300640002

RECEIVED DATA: [from Server]
Received MBAP: 00010000000701
Received PDU: 0304016300ff
WROTE DATA: - -

	Index	Value
HOLDING_REGISTERS	100	355
HOLDING_REGISTERS	101	255

Figure 24: Function 3 client message view

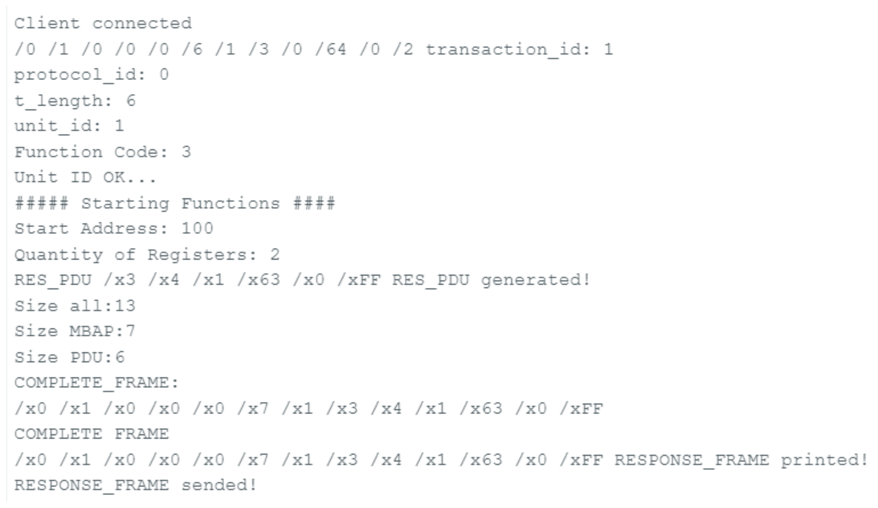
In Figure 24, As we see, we send message to server (slave) for reading analog outputs of 100th and 101st addresses. Response message to client about these addresses' outputs and result is shown.



```
//03 (0x03) Read Holding Registers
uint8_t *READ_H_REGS( uint8_t function_code, uint8_t start_address, uint8_t quantity_of_inputs)
{
  // RES_PDU nun uzunluğu 2*quantity_of_inputs (1byte*2) + byte_count(1 byte) + function_code(1byte)
  int length_res_pdu = 2*quantity_of_inputs + 1 + 1;

  //Adjustment to memory dynamically
  uint8_t *RES_PDU_3 = malloc(length_res_pdu*sizeof(uint8_t));

  uint8_t byte_count = 2*quantity_of_inputs;
  RES_PDU_3[0] = function_code;
  RES_PDU_3[1] = byte_count;

  //Burada Arduino'nun Analog Output çıkışlarını okuyacak değerler yer alacak
  for(int i = 0; i<quantity_of_inputs; i++)
  {
    RES_PDU_3[(2*i)+2] = highByte(HOLDING_REGISTERS[start_address + i]);
    RES_PDU_3[(2*i)+3] = lowByte(HOLDING_REGISTERS[start_address + i]);
  }
  Serial.print("RES_PDU ");
  for (int i = 0; i < length_res_pdu; i++)
  {
    Serial.print("/x");
    Serial.print(RES_PDU_3[i], HEX);
    Serial.print(" ");
  }
  return RES_PDU_3;
}
```

Figure 25: Function 3 Response PDU Arduino code

As we see the Figure 25, as before we first set the length of response PDU. After setting the first two bytes as function code and byte count, we created the high and low bytes of the measured analog outputs. Serial port output is shown below (Figure 26).

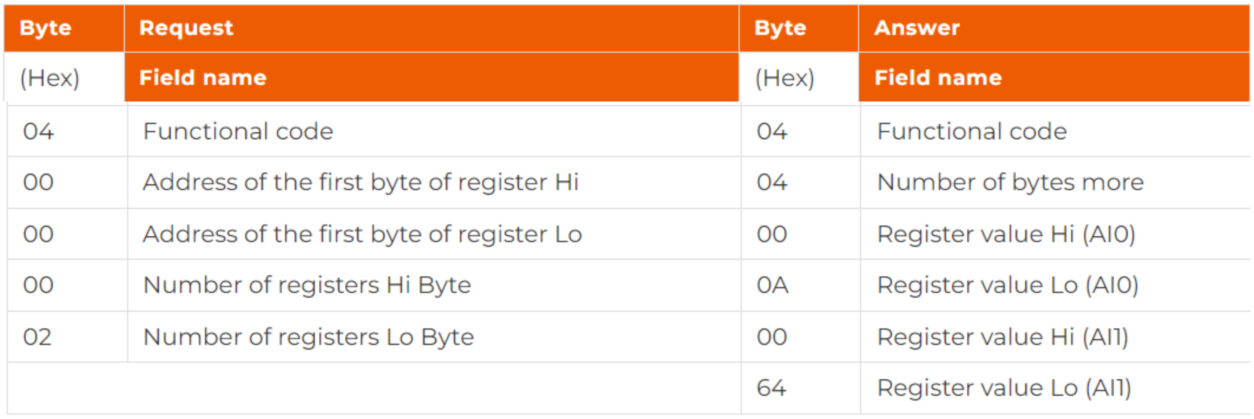


```
Client connected
/0 /1 /0 /0 /0 /6 /1 /3 /0 /64 /0 /2 transaction_id: 1
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 3
Unit ID OK...
##### Starting Functions ####
Start Address: 100
Quantity of Registers: 2
RES_PDU /x3 /x4 /x1 /x63 /x0 /xFF RES_PDU generated!
Size all:13
Size MBAP:7
Size PDU:6
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x7 /x1 /x3 /x4 /x1 /x63 /x0 /xFF
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x7 /x1 /x3 /x4 /x1 /x63 /x0 /xFF RESPONSE_FRAME printed!
RESPONSE_FRAME sended!
```

Figure 26: Function 3 Arduino Serial port output view

Function 4 Read Input Registers Arduino Code

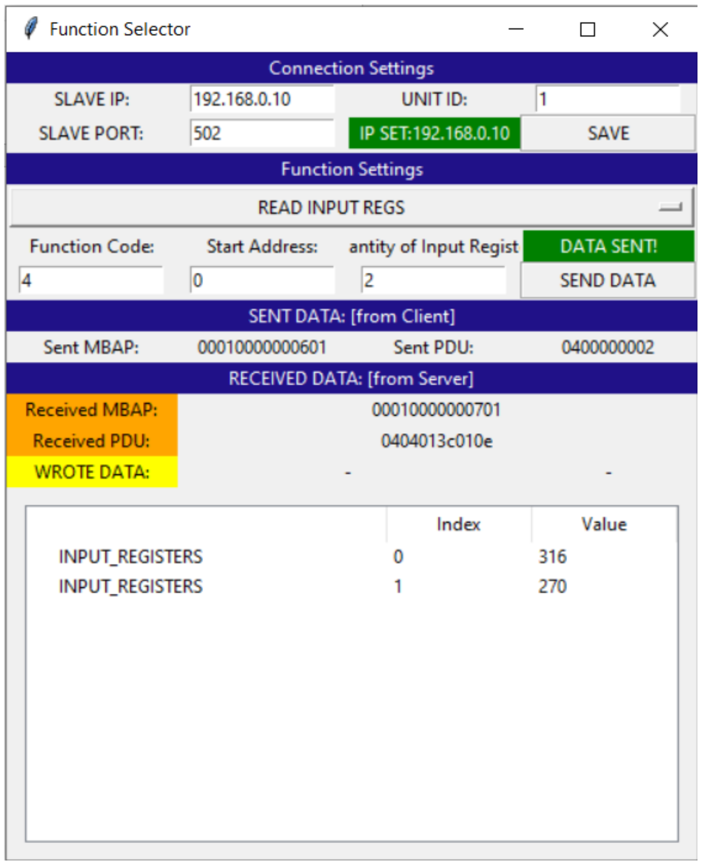
This function code is used to read from 1 to 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU, registers are addressed starting from zero.



Byte	Request	Byte	Answer
(Hex)	Field name	(Hex)	Field name
04	Functional code	04	Functional code
00	Address of the first byte of register Hi	04	Number of bytes more
00	Address of the first byte of register Lo	00	Register value Hi (AI0)
00	Number of registers Hi Byte	0A	Register value Lo (AI0)
02	Number of registers Lo Byte	00	Register value Hi (AI1)
		64	Register value Lo (AI1)

Figure 27: Example of a request to read input register 0-1

As you see the Figure 27, analog input 0's value is 10 or 0xA and analog input 1's value is 100 or 0x64 hexadecimal systems.



Function Selector

Connection Settings

SLAVE IP: 192.168.0.10 UNIT ID: 1

SLAVE PORT: 502 IP SET:192.168.0.10 SAVE

Function Settings

READ INPUT REGS

Function Code: 4 Start Address: 0 antity of Input Regist 2 DATA SENT! SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 0400000002

RECEIVED DATA: [from Server]

Received MBAP: 00010000000701

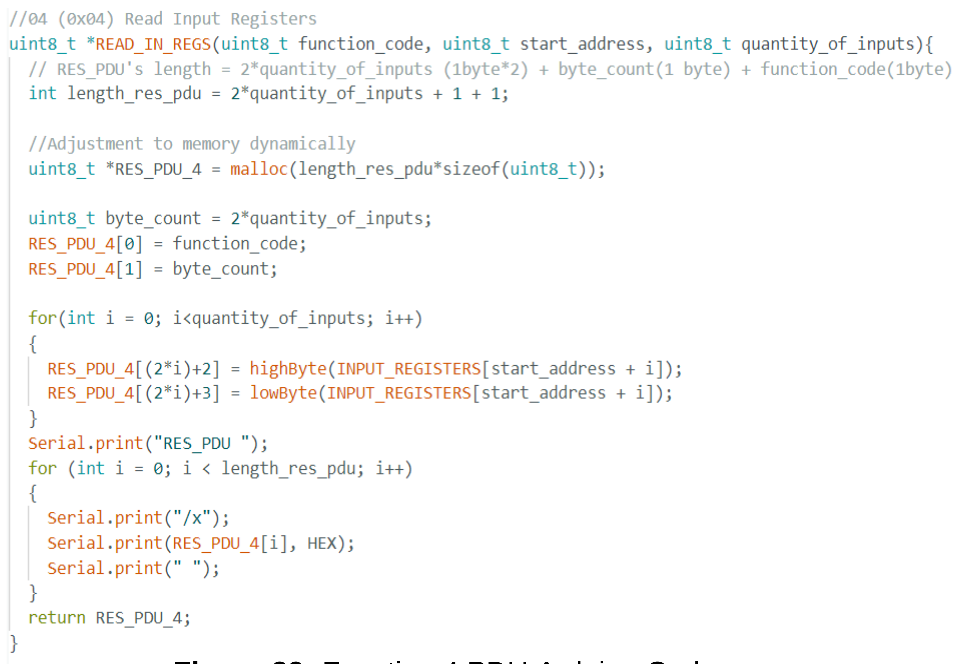
Received PDU: 0404013c010e

WROTE DATA: - -

	Index	Value
INPUT_REGISTERS	0	316
INPUT_REGISTERS	1	270

Figure 28: Function 4 client view

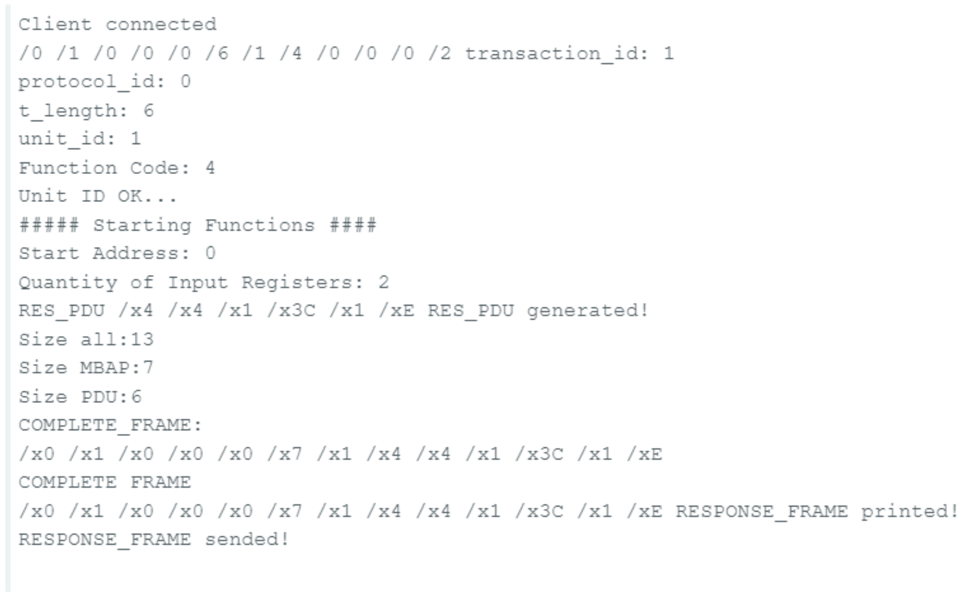
There are two potentiometers in the circuit in Figure 1. When we turn both, we can see the resulting values on the client (Figure 28).



```
//04 (0x04) Read Input Registers
uint8_t *READ_IN_REGS(uint8_t function_code, uint8_t start_address, uint8_t quantity_of_inputs){
  // RES_PDU's length = 2*quantity_of_inputs (1byte*2) + byte_count(1 byte) + function_code(1byte)
  int length_res_pdu = 2*quantity_of_inputs + 1 + 1;

  //Adjustment to memory dynamically
  uint8_t *RES_PDU_4 = malloc(length_res_pdu*sizeof(uint8_t));

  uint8_t byte_count = 2*quantity_of_inputs;
  RES_PDU_4[0] = function_code;
  RES_PDU_4[1] = byte_count;

  for(int i = 0; i<quantity_of_inputs; i++)
  {
    RES_PDU_4[(2*i)+2] = highByte(INPUT_REGISTERS[start_address + i]);
    RES_PDU_4[(2*i)+3] = lowByte(INPUT_REGISTERS[start_address + i]);
  }
  Serial.print("RES_PDU ");
  for (int i = 0; i < length_res_pdu; i++)
  {
    Serial.print("/x");
    Serial.print(RES_PDU_4[i], HEX);
    Serial.print(" ");
  }
  return RES_PDU_4;
}
```

Figure 29: Function 4 PDU Arduino Code

As in the previous functions, in function 4 we determine the length of the message from the quantity of inputs. After specifying the first two bytes as the function code and quantity of inputs, the PDU message is created (Figure 29). Arduino serial port output is shown below (Figure 30).

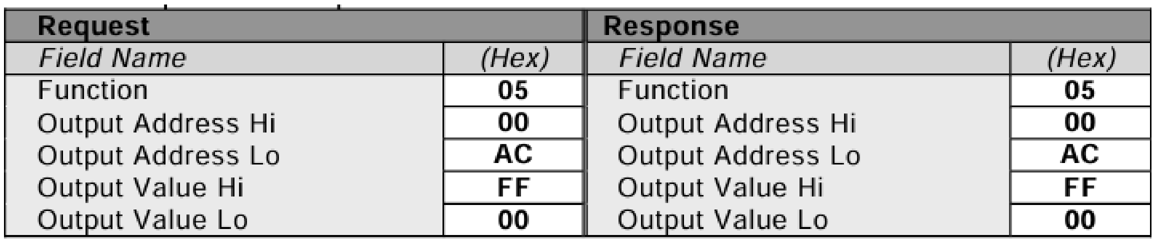


```
Client connected
/0 /1 /0 /0 /0 /6 /1 /4 /0 /0 /0 /2 transaction_id: 1
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 4
Unit ID OK...
##### Starting Functions ####
Start Address: 0
Quantity of Input Registers: 2
RES_PDU /x4 /x4 /x1 /x3C /x1 /xE RES_PDU generated!
Size all:13
Size MBAP:7
Size PDU:6
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x7 /x1 /x4 /x4 /x1 /x3C /x1 /xE
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x7 /x1 /x4 /x4 /x1 /x3C /x1 /xE RESPONSE_FRAME printed!
RESPONSE_FRAME sended!
```

Figure 30: Function 4 Arduino Serial port output view

Function 5 Write Single Coil Arduino Code

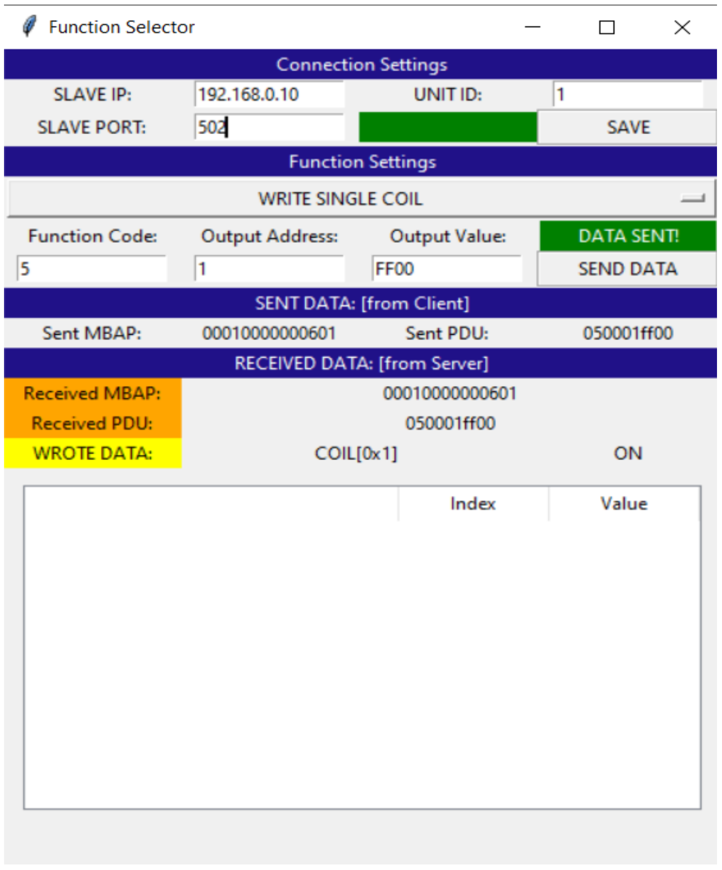
This function code is employed to write a single output to either an ON or OFF state in a remote device. The desired ON/OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the output to be ON, while a value of 00 00 requests it to be OFF. Any other values are considered illegal and will not alter the output.



Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**05**	Function	**05**
Output Address Hi	**00**	Output Address Hi	**00**
Output Address Lo	**AC**	Output Address Lo	**AC**
Output Value Hi	**FF**	Output Value Hi	**FF**
Output Value Lo	**00**	Output Value Lo	**00**

Figure 31: Example of a request to write Coil 173 ON

As seen in Figure 32, coil 173 was turned ON.



Function Selector

Connection Settings

SLAVE IP: 192.168.0.10 UNIT ID: 1

SLAVE PORT: 502

SAVE

Function Settings

WRITE SINGLE COIL

Function Code: 5

Output Address: 1

Output Value: FF00

DATA SENT!

SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 050001ff00

RECEIVED DATA: [from Server]

Received MBAP: 00010000000601

Received PDU: 050001ff00

WROTE DATA: COIL[0x1] ON

Index	Value

Figure 32: Function 5 client view

When the client sends 0xFF00 to register 1, led 1 is turned on (Figure 33).

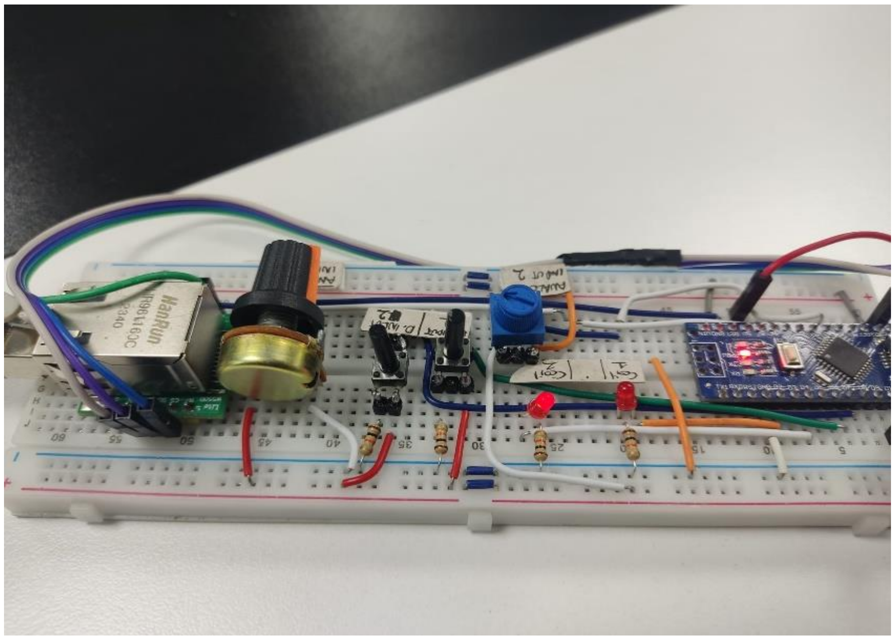
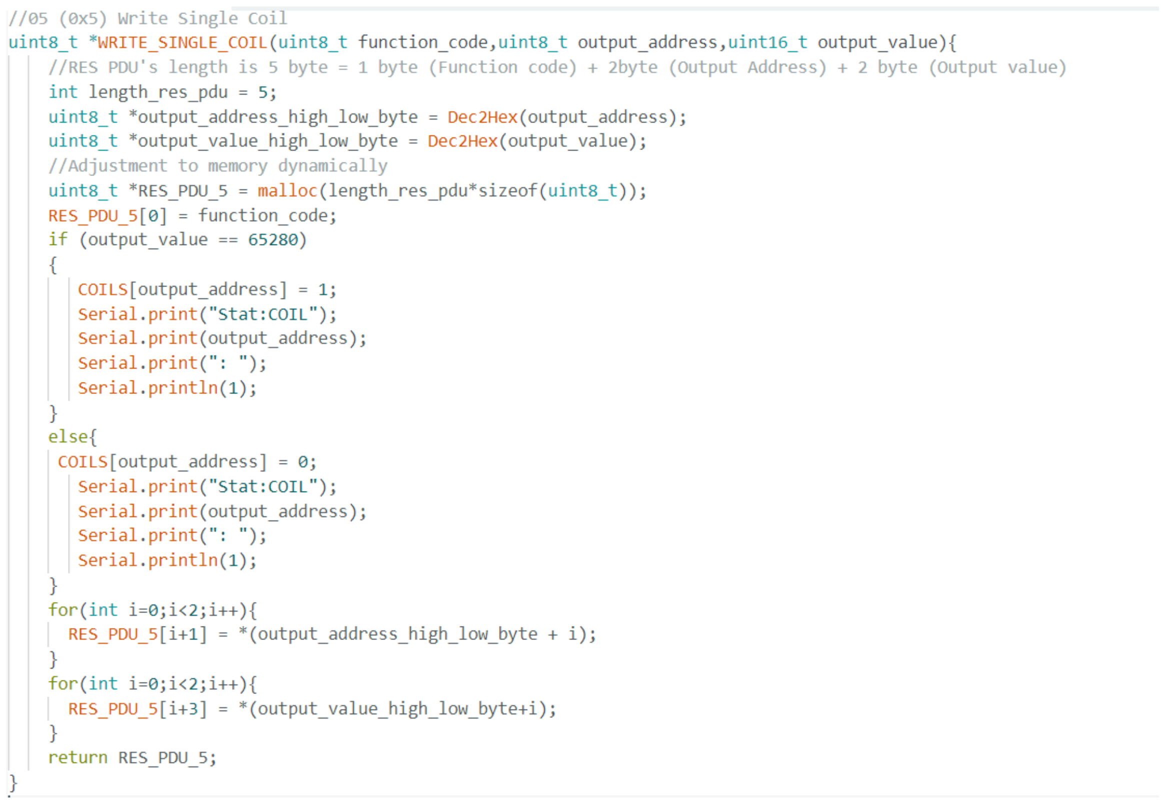


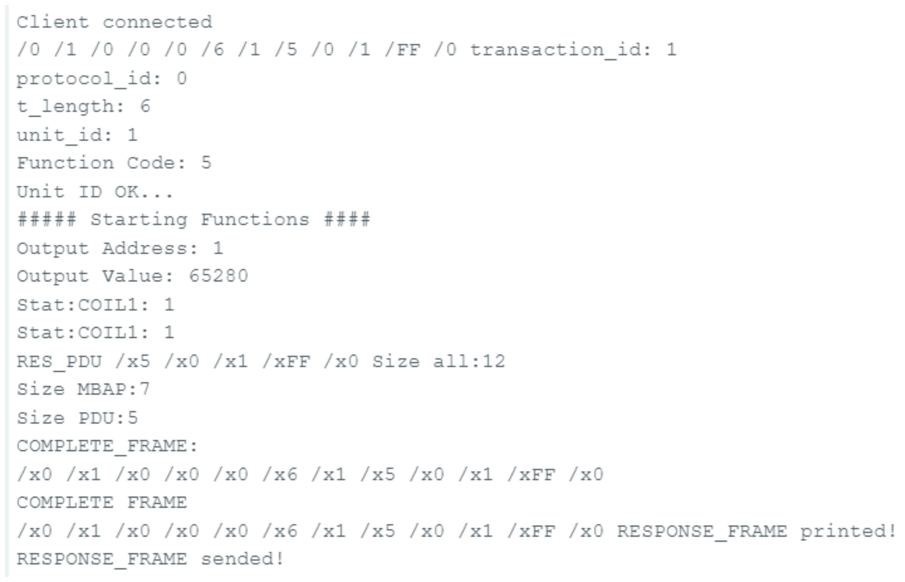
Figure 33: Turning led with function 5's 0xFF00 value



```
//05 (0x5) Write Single Coil
uint8_t *WRITE_SINGLE_COIL(uint8_t function_code,uint8_t output_address,uint16_t output_value){
    //RES PDU's length is 5 byte = 1 byte (Function code) + 2byte (Output Address) + 2 byte (Output value)
    int length_res_pdu = 5;
    uint8_t *output_address_high_low_byte = Dec2Hex(output_address);
    uint8_t *output_value_high_low_byte = Dec2Hex(output_value);
    //Adjustment to memory dynamically
    uint8_t *RES_PDU_5 = malloc(length_res_pdu*sizeof(uint8_t));
    RES_PDU_5[0] = function_code;
    if (output_value == 65280)
    {
       COILS[output_address] = 1;
       Serial.print("Stat:COIL");
       Serial.print(output_address);
       Serial.print(": ");
       Serial.println(1);
    }
    else{
     COILS[output_address] = 0;
       Serial.print("Stat:COIL");
       Serial.print(output_address);
       Serial.print(": ");
       Serial.println(1);
    }
    for(int i=0;i<2;i++){
      RES_PDU_5[i+1] = *(output_address_high_low_byte + i);
    }
    for(int i=0;i<2;i++){
      RES_PDU_5[i+3] = *(output_value_high_low_byte+i);
    }
    return RES_PDU_5;
}
```

Figure 34. Function 5 Response PDU Arduino Code

Since the length of the PDU response message in Function 5 is fixed, we set it to 5 bytes. If the output value is equal to 65280 (0xFF00), turn on the coil at the output address. If the client returns 0, the coil turns off commands at the output address are applied (Figure 34). Arduino Serial port output is shown below (Figure 35).

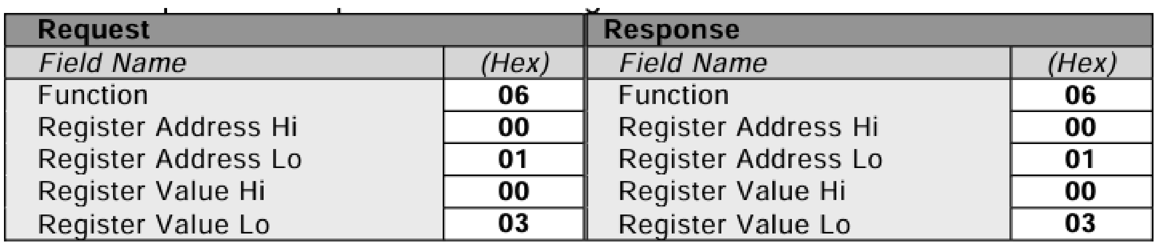


```
Client connected
/0 /1 /0 /0 /0 /6 /1 /5 /0 /1 /FF /0 transaction_id: 1
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 5
Unit ID OK...
##### Starting Functions ####
Output Address: 1
Output Value: 65280
Stat:COIL1: 1
Stat:COIL1: 1
RES_PDU /x5 /x0 /x1 /xFF /x0 Size all:12
Size MBAP:7
Size PDU:5
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x6 /x1 /x5 /x0 /x1 /xFF /x0
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x6 /x1 /x5 /x0 /x1 /xFF /x0 RESPONSE_FRAME printed!
RESPONSE_FRAME sended!
```

Figure 35. Arduino Serial Port Output View

Function 6 Write Single Register Arduino Code

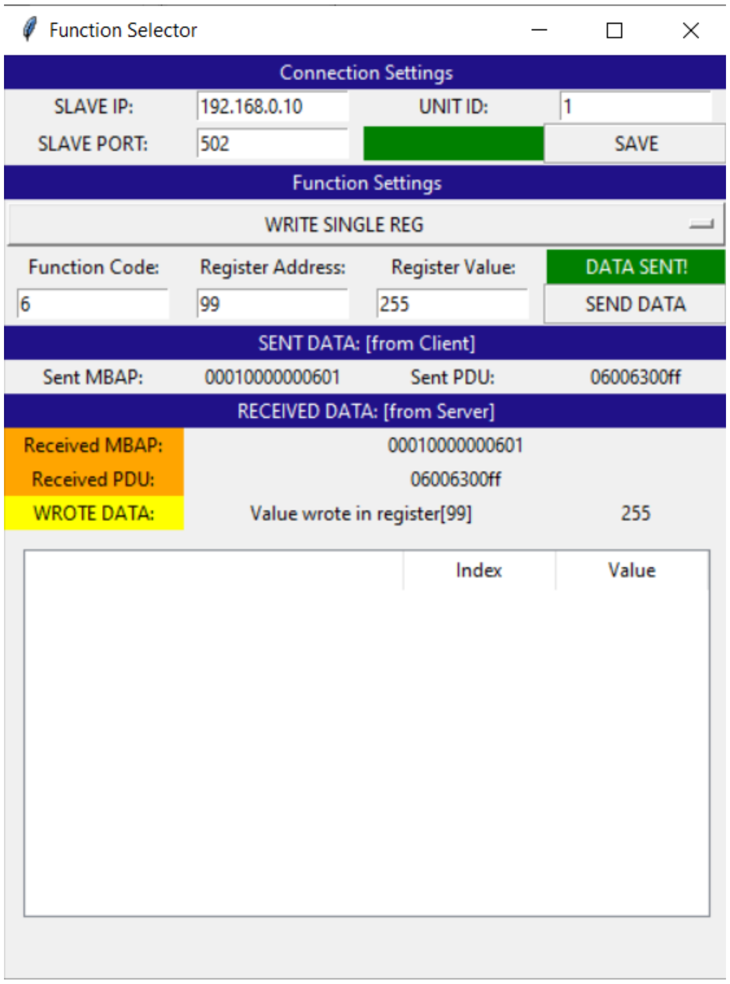
This function code is used to write a single holding register in a remote device. The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore, register numbered 1 is addressed as 0.



Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**06**	Function	**06**
Register Address Hi	**00**	Register Address Hi	**00**
Register Address Lo	**01**	Register Address Lo	**01**
Register Value Hi	**00**	Register Value Hi	**00**
Register Value Lo	**03**	Register Value Lo	**03**

Figure 36: example of a request to write register 2 to 00 03 hex

In Figure 36, we see that the value 3 is written in register 2.


Function Selector

Connection Settings

SLAVE IP:	192.168.0.10	UNIT ID:	1
SLAVE PORT:	502	🟩	SAVE

Function Settings

WRITE SINGLE REG

Function Code:	Register Address:	Register Value:	DATA SENT!
6	99	255	SEND DATA

SENT DATA: [from Client]

Sent MBAP:	00010000000601	Sent PDU:	06006300ff

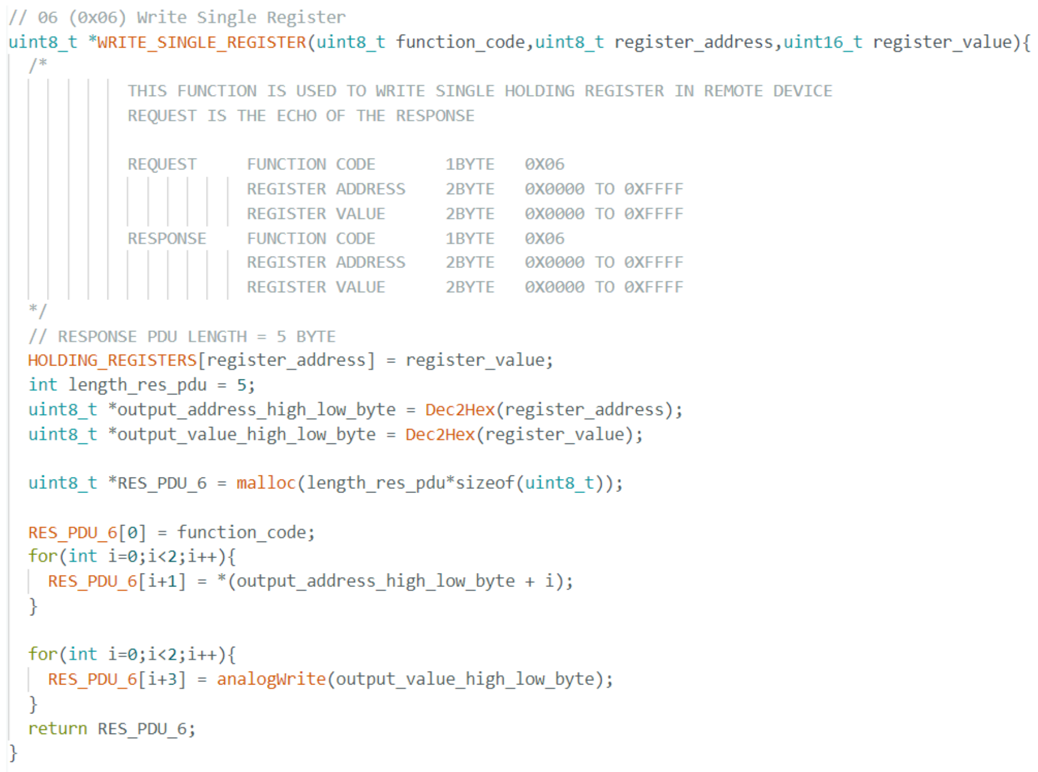
RECEIVED DATA: [from Server]

Received MBAP:	00010000000601	
Received PDU:	06006300ff	
WROTE DATA:	Value wrote in register[99]	255

	Index	Value

Figure 37: Function 6 client view

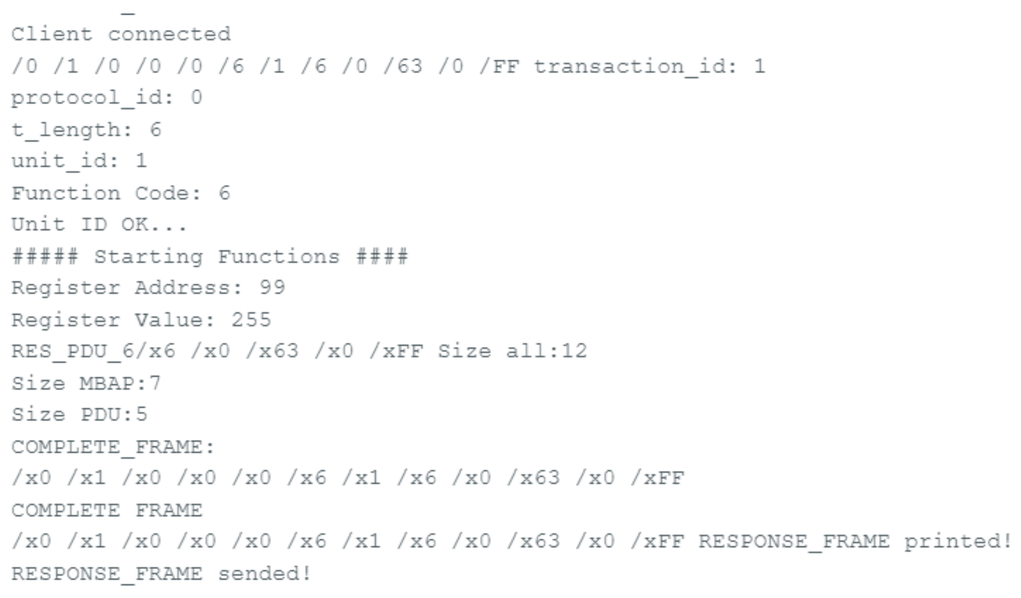
When we write the register value to the register address specified in the client, we see that the value is successfully transferred to the relevant register in the GUI (Figure 37).



```
// 06 (0x06) Write Single Register
uint8_t *WRITE_SINGLE_REGISTER(uint8_t function_code,uint8_t register_address,uint16_t register_value){
  /*
          THIS FUNCTION IS USED TO WRITE SINGLE HOLDING REGISTER IN REMOTE DEVICE
          REQUEST IS THE ECHO OF THE RESPONSE

          REQUEST     FUNCTION CODE       1BYTE   0X06
                      REGISTER ADDRESS    2BYTE   0X0000 TO 0XFFFF
                      REGISTER VALUE      2BYTE   0X0000 TO 0XFFFF
          RESPONSE    FUNCTION CODE       1BYTE   0X06
                      REGISTER ADDRESS    2BYTE   0X0000 TO 0XFFFF
                      REGISTER VALUE      2BYTE   0X0000 TO 0XFFFF
  */
  // RESPONSE PDU LENGTH = 5 BYTE
  HOLDING_REGISTERS[register_address] = register_value;
  int length_res_pdu = 5;
  uint8_t *output_address_high_low_byte = Dec2Hex(register_address);
  uint8_t *output_value_high_low_byte = Dec2Hex(register_value);

  uint8_t *RES_PDU_6 = malloc(length_res_pdu*sizeof(uint8_t));

  RES_PDU_6[0] = function_code;
  for(int i=0;i<2;i++){
    RES_PDU_6[i+1] = *(output_address_high_low_byte + i);
  }

  for(int i=0;i<2;i++){
    RES_PDU_6[i+3] = analogWrite(output_value_high_low_byte);
  }
  return RES_PDU_6;
}
```

Figure 38: Function 6 Response PDU Arduino Code

In Function 6, we directly send the message from the client to the relevant register value. After separating the output value and address into high and low bytes, the created response message is sent to the client (Figure 38). Arduino serial monitor output is shown below (Figure 39):



```
Client connected
/0 /1 /0 /0 /0 /6 /1 /6 /0 /63 /0 /FF transaction_id: 1
protocol_id: 0
t_length: 6
unit_id: 1
Function Code: 6
Unit ID OK...
##### Starting Functions ####
Register Address: 99
Register Value: 255
RES_PDU_6/x6 /x0 /x63 /x0 /xFF Size all:12
Size MBAP:7
Size PDU:5
COMPLETE_FRAME:
/x0 /x1 /x0 /x0 /x0 /x6 /x1 /x6 /x0 /x63 /x0 /xFF
COMPLETE FRAME
/x0 /x1 /x0 /x0 /x0 /x6 /x1 /x6 /x0 /x63 /x0 /xFF RESPONSE_FRAME printed!
RESPONSE_FRAME sended!
```

Figure 39: Arduino Serial Monitor Output view

ENGINEERING STANDARDS and DESIGN CONSTRAINTS

Economy:

Since our budget is limited we need to use cost-effective microcontrollers and communication tools. Special thanks to our advisor, Ergün Gözek, who provides Schneider Electric M221 PLC to test our Master program.

Manufacturability:

Our design should not to be dependent on pre-made libraries. Although we start using Arduino’s Ethernet library at the beginning of the project, we moved on the W5500 chips libraries which is provided from the manufacturer. Therefore we need to write our own adaptation code to use manufacturers libraries which is free of charge.

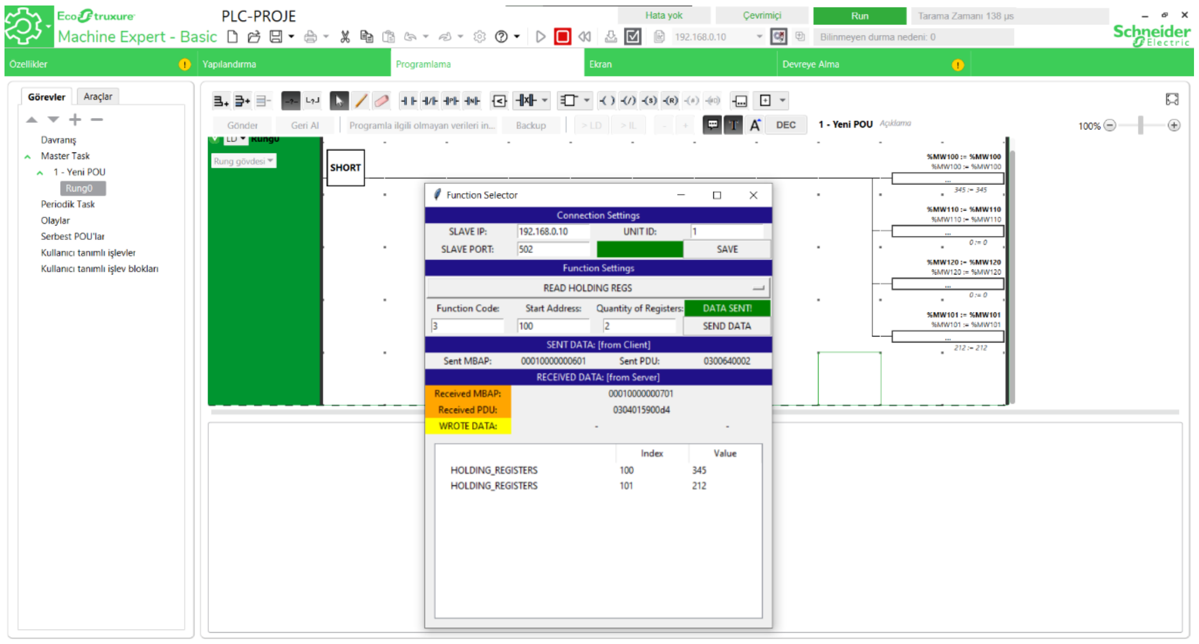
RESULTS AND DISCUSSIONS

Result of communication is shown in the proposed solution section of the report for every implemented MODBUS function. This section briefly explain the communication results in three different cases.

- Python Master – Schneider M221 PLC Slave
- Python Master – Arduino Nano Slave
- Python Master – STM32F407 Discovery Board Slave

Python Master – Schneider M221 PLC Slave

In this test configuration, read holding registers function is used in MODBUS TCP/IP communication between M221 PLC and our Master python program. As it can be seen from the figure X, holding registers 100 and 101 read from the PLC. Their value is shown in Master program's result section. Also, write single register function works as expected, but the screenshot is not included in this section. This results proves that MODBUS TCP/IP communication is established between our Master python program and commercial slave PLC.



Connection Settings

SLAVE IP: 192.168.0.10 UNIT ID: 1

SLAVE PORT: 502 SAVE

Function Settings

READ HOLDING REGS

Function Code:	Start Address:	Quantity of Registers:	DATA SENT!
3	100	2	SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 0300640002

RECEIVED DATA: [from Server]

Received MBAP: 00010000000701

Received PDU: 0304015900d4

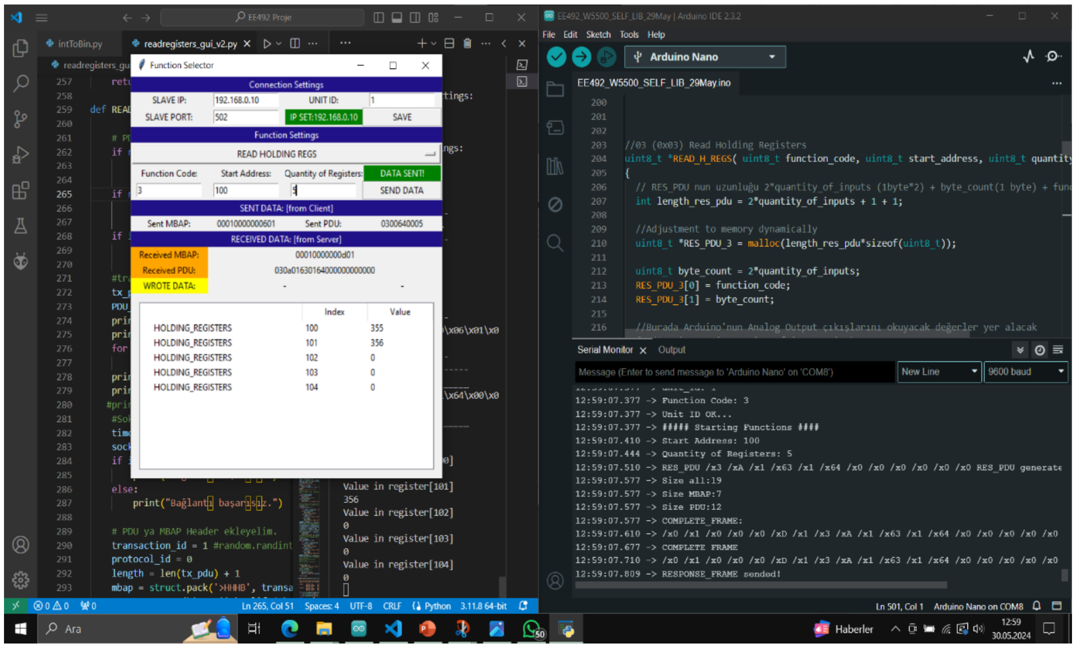
WROTE DATA: - -

	Index	Value
HOLDING_REGISTERS	100	345
HOLDING_REGISTERS	101	212

Figure 40. MODBUS communication among Python Master and M221 PLC Slave

Python Master – Arduino Nano Slave

This test case is shown for every function in proposed solution part. Therefore, only the same function with test 1, which is read holding registers is shown in the figure below.



Function Selector

Connection Settings			
SLAVE IP:	192.168.0.10	UNIT ID:	1
SLAVE PORT:	502	IP SET:192.168.0.10	SAVE

Function Settings

READ HOLDING REGS

Function Code:	Start Address:	Quantity of Registers:	DATA SENT!
3	100	5	SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 0300640005

RECEIVED DATA: [from Server]

Received MBAP: 00010000000d01
Received PDU: 030a016301640000000000000
WROTE DATA: - -

	Index	Value
HOLDING_REGISTERS	100	355
HOLDING_REGISTERS	101	356
HOLDING_REGISTERS	102	0
HOLDING_REGISTERS	103	0
HOLDING_REGISTERS	104	0

```python
    else:
        print("Bağlantı başarısız.")

    # PDU ya MBAP Header ekleyelim.
    transaction_id = 1 #random.randint
    protocol_id = 0
    length = len(tx_pdu) + 1
    mbap = struct.pack('>HHHB', transa
```

```
Value in register[101]
356
Value in register[102]
0
Value in register[103]
0
Value in register[104]
0
```

EE492_W5500_SELF_LIB_29May.ino

```c
//03 (0x03) Read Holding Registers
uint8_t *READ_H_REGS( uint8_t function_code, uint8_t start_address, uint8_t quantity
{
  // RES_PDU nun uzunluğu 2*quantity_of_inputs (1byte*2) + byte_count(1 byte) + func
  int length_res_pdu = 2*quantity_of_inputs + 1 + 1;

  //Adjustment to memory dynamically
  uint8_t *RES_PDU_3 = malloc(length_res_pdu*sizeof(uint8_t));

  uint8_t byte_count = 2*quantity_of_inputs;
  RES_PDU_3[0] = function_code;
  RES_PDU_3[1] = byte_count;

  //Burada Arduino'nun Analog Output çıkışlarını okuyacak değerler yer alacak
```

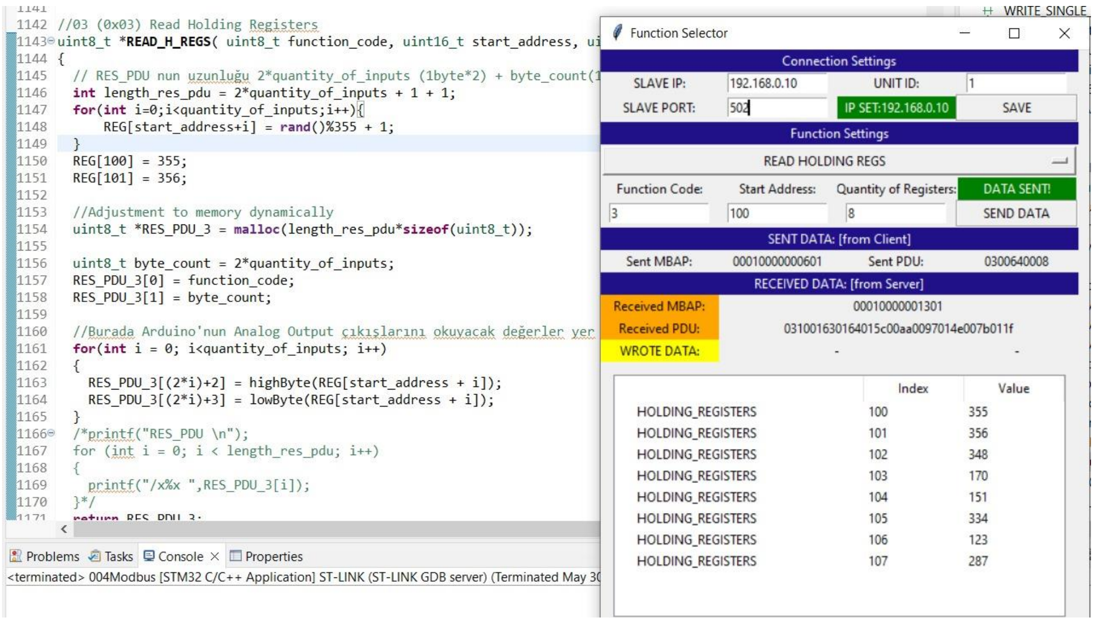
Serial Monitor — Message (Enter to send message to 'Arduino Nano' on 'COM8') — New Line — 9600 baud

```
12:59:07.377 -> Function Code: 3
12:59:07.377 -> Unit ID OK...
12:59:07.377 -> ##### Starting Functions ####
12:59:07.410 -> Start Address: 100
12:59:07.444 -> Quantity of Registers: 5
12:59:07.510 -> RES_PDU /x3 /xA /x1 /x63 /x1 /x64 /x0 /x0 /x0 /x0 /x0 /x0 RES_PDU generate
12:59:07.577 -> Size all:19
12:59:07.577 -> Size MBAP:7
12:59:07.577 -> Size PDU:12
12:59:07.577 -> COMPLETE_FRAME:
12:59:07.610 -> /x0 /x1 /x0 /x0 /x0 /xD /x1 /x3 /xA /x1 /x63 /x1 /x64 /x0 /x0 /x0 /x0 /x0
12:59:07.677 -> COMPLETE FRAME
12:59:07.710 -> /x0 /x1 /x0 /x0 /x0 /xD /x1 /x3 /xA /x1 /x63 /x1 /x64 /x0 /x0 /x0 /x0 /x0
12:59:07.809 -> RESPONSE_FRAME sended!
```

Figure 41. Python Master, Arduino Nano Slave.

Python Master – STM32F407 Discovery Board Slave

Since tests in Arduino Nano is successful, we switched from 8-bit microcontroller based board to 32-bit microcontroller based board which is STM32F4 discovery board. Here's the same function output can be seen in figure 42.



```c
//03 (0x03) Read Holding Registers
uint8_t *READ_H_REGS( uint8_t function_code, uint16_t start_address, ui
{
  // RES_PDU nun uzunluğu 2*quantity_of_inputs (1byte*2) + byte_count(1
  int length_res_pdu = 2*quantity_of_inputs + 1 + 1;
  for(int i=0;i<quantity_of_inputs;i++){
      REG[start_address+i] = rand()%355 + 1;
  }
  REG[100] = 355;
  REG[101] = 356;

  //Adjustment to memory dynamically
  uint8_t *RES_PDU_3 = malloc(length_res_pdu*sizeof(uint8_t));

  uint8_t byte_count = 2*quantity_of_inputs;
  RES_PDU_3[0] = function_code;
  RES_PDU_3[1] = byte_count;

  //Burada Arduino'nun Analog Output çıkışlarını okuyacak değerler yer
  for(int i = 0; i<quantity_of_inputs; i++)
  {
    RES_PDU_3[(2*i)+2] = highByte(REG[start_address + i]);
    RES_PDU_3[(2*i)+3] = lowByte(REG[start_address + i]);
  }
  /*printf("RES_PDU \n");
  for (int i = 0; i < length_res_pdu; i++)
  {
    printf("/x%x ",RES_PDU_3[i]);
  }*/
```

Problems Tasks Console Properties

```
<terminated> 004Modbus [STM32 C/C++ Application] ST-LINK (ST-LINK GDB server) (Terminated May 30
```

Function Selector

Connection Settings			
SLAVE IP:	192.168.0.10	UNIT ID:	1
SLAVE PORT:	502	IP SET:192.168.0.10	SAVE

Function Settings

READ HOLDING REGS

Function Code:	Start Address:	Quantity of Registers:	DATA SENT!
3	100	8	SEND DATA

SENT DATA: [from Client]

Sent MBAP: 00010000000601 Sent PDU: 0300640008

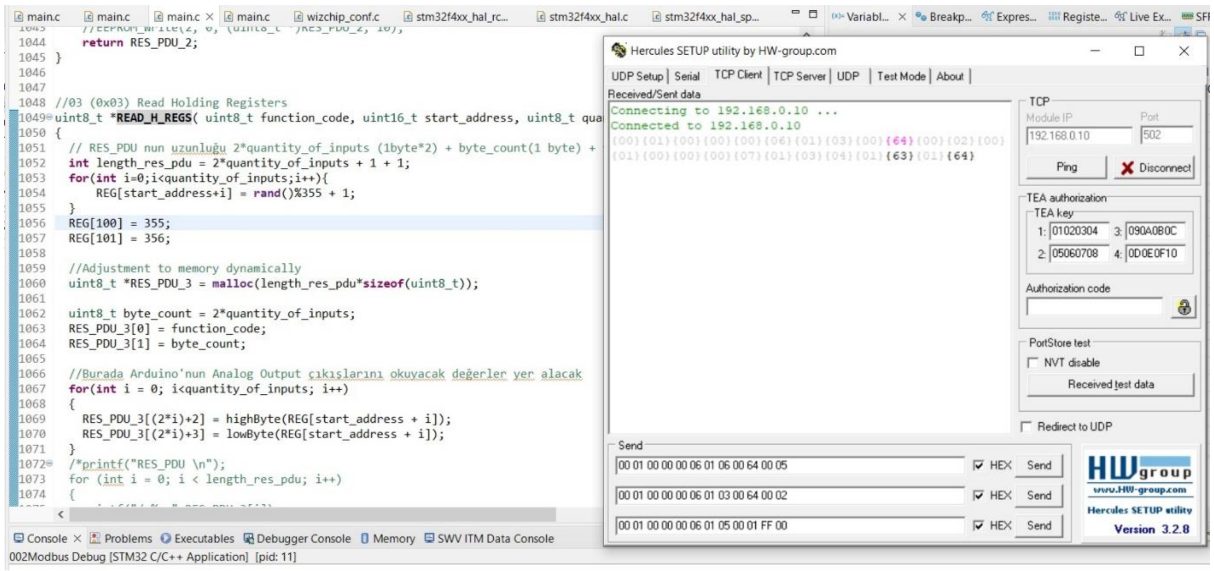
RECEIVED DATA: [from Server]

Received MBAP: 00010000001301
Received PDU: 031001630164015c00aa0097014e007b011f
WROTE DATA: - -

	Index	Value
HOLDING_REGISTERS	100	355
HOLDING_REGISTERS	101	356
HOLDING_REGISTERS	102	348
HOLDING_REGISTERS	103	170
HOLDING_REGISTERS	104	151
HOLDING_REGISTERS	105	334
HOLDING_REGISTERS	106	123
HOLDING_REGISTERS	107	287

Figure 42. Python Master, STM32F4 Discovery Slave.

However, an error occurred in our test in STM32F4 Discovery board. The Discovery board does not respond more than one request from our python master. The problem might be at the socket open and close operations of the slave device. In order to satisfy that discovery board is working, several tests applied in another communication software called Hercules. All implemented functions worked in tests with Hercules program. Therefore, the problem is not in our functions and implementation of the protocol, but in the TCP communication through W5500 chip between python master and discovery slave. The resulting test output is demonstrated in figure 43.



```
    return RES_PDU_2;
}


//03 (0x03) Read Holding Registers
uint8_t *READ_H_REGS( uint8_t function_code, uint16_t start_address, uint8_t qua
{
  // RES_PDU nun uzunluğu 2*quantity_of_inputs (1byte*2) + byte_count(1 byte) +
  int length_res_pdu = 2*quantity_of_inputs + 1 + 1;
  for(int i=0;i<quantity_of_inputs;i++){
      REG[start_address+i] = rand()%355 + 1;
  }
  REG[100] = 355;
  REG[101] = 356;

  //Adjustment to memory dynamically
  uint8_t *RES_PDU_3 = malloc(length_res_pdu*sizeof(uint8_t));

  uint8_t byte_count = 2*quantity_of_inputs;
  RES_PDU_3[0] = function_code;
  RES_PDU_3[1] = byte_count;

  //Burada Arduino'nun Analog Output çıkışlarını okuyacak değerler yer alacak
  for(int i = 0; i<quantity_of_inputs; i++)
  {
    RES_PDU_3[(2*i)+2] = highByte(REG[start_address + i]);
    RES_PDU_3[(2*i)+3] = lowByte(REG[start_address + i]);
  }
  /*printf("RES_PDU \n");
  for (int i = 0; i < length_res_pdu; i++)
  {
```

```
002Modbus Debug [STM32 C/C++ Application]  [pid: 11]
```

Hercules SETUP utility by HW-group.com

Received/Sent data

```
Connecting to 192.168.0.10 ...
Connected to 192.168.0.10
{00}{01}{00}{00}{00}{06}{01}{03}{00}{64}{00}{02}{00}
{01}{00}{00}{00}{07}{01}{03}{04}{01}{63}{01}{64}
```

TCP Module IP: 192.168.0.10 Port: 502

TEA key 1: 01020304 2: 05060708 3: 090A0B0C 4: 0D0E0F10

Send

```
00 01 00 00 00 06 01 06 00 64 00 05
00 01 00 00 00 06 01 03 00 64 00 02
00 01 00 00 00 06 01 05 00 01 FF 00
```

Figure 43. Hercules Master, STM32F4 Discovery Slave.

In this test case, the input TCP frame's bytes are hardcoded to Hercules program interface. The response bytes can be seen in received/sent data section. Hexadecimal value of 0x0163 corresponds to decimal value of 355, 0x0164 corresponds to decimal value of 356. By this way, it is shown that communication is established and function 3 is performed.

To summarize, in all cases although there are minor faults, majority of MODBUS TCP/IP communication functions between Master program and Slave programs is implemented successfully. Our Master communicated with commercial PLC and our embedded systems.

CONCLUSIONS

With the MODBUS communication protocol, the primary objective is to facilitate data exchange between the master and slave devices by utilizing the functions defined within the protocol. In our project, the Python interface we developed on the computer was designated as the master. To create the slave device, we established a circuit and enabled communication through the use of microcontrollers. Although not all functions were implemented, the first six functions were successfully completed, demonstrating effective data exchange between an 8-bit Arduino Nano and the Python interface. This process involved receiving data, processing it, and then sending it back to the interface. Furthermore, we extended our testing to a more advanced microcontroller, specifically the 32-bit STM32F4. This microcontroller showed successful operation not only in our custom-built systems but also in pre-existing systems such as Hercule. While the Python interface program we designed successfully communicated with the STM32F4, it only managed to establish communication once. This limitation led us to hypothesize that the error could stem from several factors: a potential flaw in the written algorithm, technical incompatibility between the interface and the STM32F4, or a frequency mismatch related to the operating speed of the STM32F4. Given these challenges, it is believed that further testing on different 32-bit microcontrollers may provide deeper insights into the root cause of the communication issue. Addressing these incompatibilities is crucial for improving the robustness of the system.

To elevate the project to a higher level of functionality and reliability, several enhancements are proposed. One key suggestion is to transition from using a breadboard to designing and implementing a printed circuit board (PCB). This change would likely increase the stability and durability of the hardware setup. Additionally, further refinement and enhancement of the interface program are essential. These improvements would ensure more reliable and consistent communication between the master and slave devices, thereby advancing the overall effectiveness and efficiency of the system.

REFERENCES

[1] modbus.org. (n.d.). Modbus. Retrieved from https://www.modbus.org/

[2] Modbus-IDA. (n.d.). Modbus Application Protocol V1.1b. Retrieved from https://modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

[3] IPC2U. (n.d.). Detailed Description of the Modbus TCP Protocol with Command Examples. Retrieved from https://ipc2u.com/articles/knowledge-base/detailed-description-of-the-modbus-tcp-protocol-with-command-examples/

[4] Modbus-IDA. (n.d.). Modbus Messaging Implementation Guide V1.0b. Retrieved from https://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf