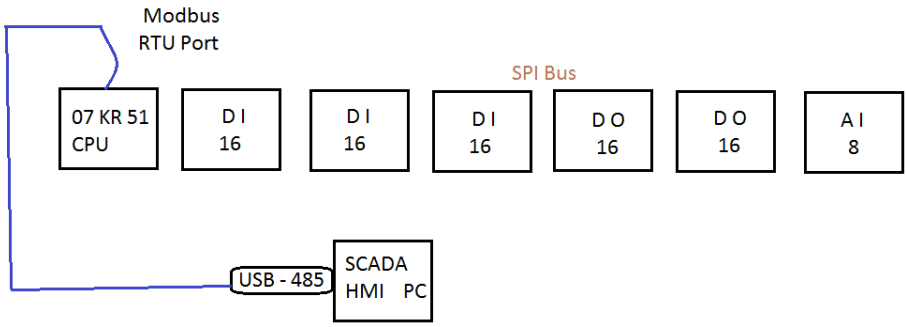


Request – Response System

ASCII = all ASCII character permitted

RTU = Hex = only 0 – 9 A – F charecters are allowed . All exchanged values must be In hex

RTU frame :-

Addr	Func	Count	Data	Data	Data	Data	CR C_{Lo}	CR C_{Hi}

Data Type		Base Address	PLC	Size
Input Register	Read Only	30001	Analog Input	16 BIts
Coil	Read / Write	00001	Digital Output	1 Bit
Input Descrete	Read Only	10001	Digital Input	1 Bit
Out Register Holding	Read / Write	40001	Analog Out	16 Bits

Request

==> Get analog output holding registers # 40108 to 40110 from device 17

START ADDRESS = ADD OF 1ST REG = HEX OF 108 NOT HEX OF 40108

Address of the first register requested ==> 40108 - 40001 = 107 Dec = 6B hex

DeviceID Opcode StartAddress Count CRC_L CRC_H

11 03 006B 0003 0000

11 03 006B 0003 76 87

11: The Slave Address (17 = 11 hex)

03: The Function Code (read Analog Output Holding Registers)

006B: The Data Address of the first register requested. (40108-40001 = 107 = 6B hex)

0003: The total number of registers requested. (read 3 registers 40108 to 40110)

76 87: The CRC

Response

11 03 06 AE41 5652 4340 49AD

11: The Slave Address (17 = 11 hex)

03: The Function Code (read Analog Output Holding Registers)

06: The number of data bytes to follow (3 registers x 2 bytes each = 6 bytes)

AE41: The contents of register 40108

5652: The contents of register 40109

4340: The contents of register 40110

49AD: The CRC

The Modbus protocol is a single master protocol.Slaves are allowed to send telegrams only if master asks. The MODBUS defines a silent-interval of at least 3.5 chars between two telegrams.

Modbus registers are transmitted with the high address byte first followed by the low byte. = Big Endian

32 Bit floats are in accordance with IEEE 754 → Low Reg-16 is on left side but Low Byte is on Right

If you want to read register 40001 (from memory map of device) the address in the code will be 0
RESPONSE - READING HOLDING REG. START = 108 (Coded as 107). NO = 4

Index	Response	
0	Slave address	1 Byte
1	Function Returned	1 Byte
2	Byte Count	1 Byte
3	Reg 4018 – High Byte LS Byte First	1 Byte
4	Reg 4018 – Low Byte	1 Byte
5	Reg 4019 – High Byte	1 Byte
6	Reg 4019 – Low Byte	1 Byte
7	Reg 4020 – High Byte	1 Byte
8	Reg 4020 – Low Byte	1 Byte
9	Reg 4021 – High Byte	1 Byte
10	Reg 4021 – Low Byte	1 Byte
	CRC – LOW Byte High Byte First	1 Byte
	CRC – HIGH Byte	1 Byte

In a normal response, the slave echoes the function code. The first sign of an exception response is that the function code returned with its highest bit set. All function codes have 0 for their most significant bit. Therefore, setting this bit to 1 is the signal that the slave cannot process the request.. Error response is 5 bytes long

Exception Response

Request :-
This command is requesting the ON/OFF status of discrete coil #1186 from the slave device with address 10.
0A 01 04A1 0001 AC 63
0A: The Slave Address 10
01: The Function Code (read Coil Status)
04A1: The Data Address of the first coil to read. (Coil 1186 - 1 = 1185)
04A1 hex
0001: The total number of coils requested.
AC63: The CRC
Response :-
0A 81 02 B053
0A: The Slave Address (10 = 0A hex)
81: The Function Code (read Coil Status - with the highest bit set)
02: The Exception Code
B0 CRC Low
53 CRC High

Example: Request a float number(32-Bit) on register addresses 40108 and 40109 of device 17
The Data Address of the first register => 40108 - 40001 = 107 = 6B hex

Master -> Slave 11 03 00 6B 00 02 crc_L crc_H
Slave -> Master 11 03 04 CC CD 42 8D crc_L crc_H

4 = No of **Data** Bytes returned
4 in Holding register is to show that its holding and its not a part in the code
address 40108 = register 108 ===> coded as 107 = 00 6B
Device to PC = Big End to Little End = Reverse Bytes / Word

CC CD 42 8D (Big End of Device)
A B C D
Make Word by reversing then make Float by reversing again
8D 42 CD CC
D C B A

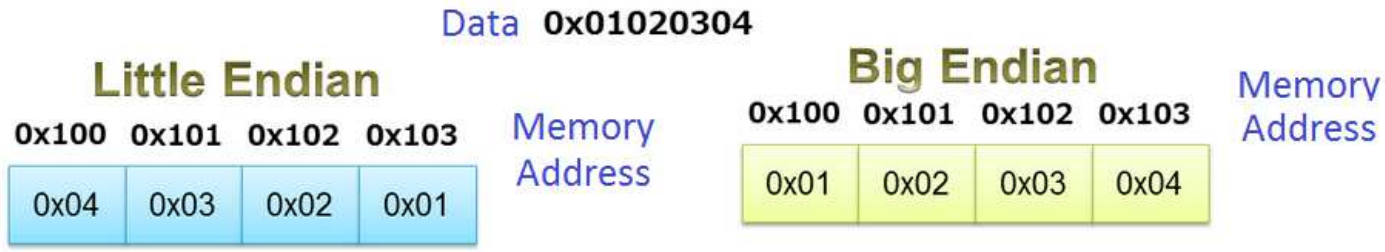
Function Codes

01 Read Coil Status = Read DO current value
02 Read Input Status = Read DI
03 Read Holding Registers = Read AO reg
04 Read Input Registers = Read AI reg
05 Force Single Coil = Digital Out
06 Preset Single Register = Write into register / AO
15 Force Multiple Coils = set on / off multiple DO at once
16 Preset Multiple Registers = set value in AO / reg at once
23 Read and Write 4X holding Registers

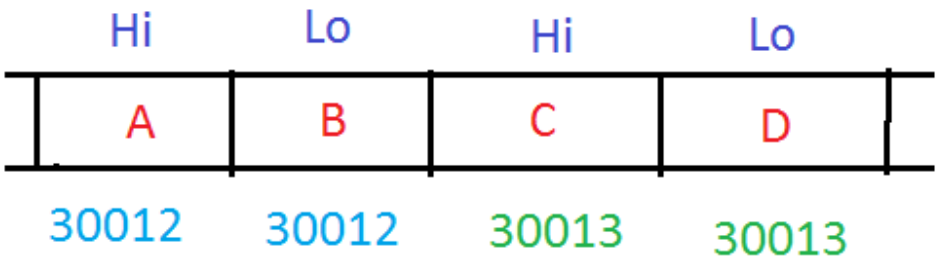
Endianess

Little Endian = Small memory has Small Byte (Least Significant Byte)
and big memory address has big Byte (Most Significant Byte)
Memory < -- > Byte Same
Intel Architecture

Big Endian = Small memory has big Byte (Most Significant Byte)
and big memory address has Small Byte (Least Significant Byte)
Memory < -- > Byte Opposite
Motoroal Architecture
Modbus Uses Big Endian Architecture



Modbus uses BIG Endian (Motorola PC)way of saving data inside its memory and Intel uses Little Endian Architecture.



C# Byte[] = { B , A , D , C } <-- Little Endian PC

```
byte[] x_100 = BitConverter.GetBytes(0xD5EA);
byte[] x_101 = BitConverter.GetBytes(0x436F);
byte[] b = new byte[] { x_100[0], x_100[1], x_101[0], x_101[1] };
float f = BitConverter.ToSingle(b, 0);  // C# floats are IEEE 754 already
MessageBox.Show(f.ToString());
```

==> hex from int

```
int Value = 182;
string hex_Value = Value.ToString("X");
string hex_Value = Value.ToString("X2");    ==> F is returned as 0F
```

==> int from hex

```
int Value = int.Parse(hex_value, System.Globalization.NumberStyles.HexNumber)
```

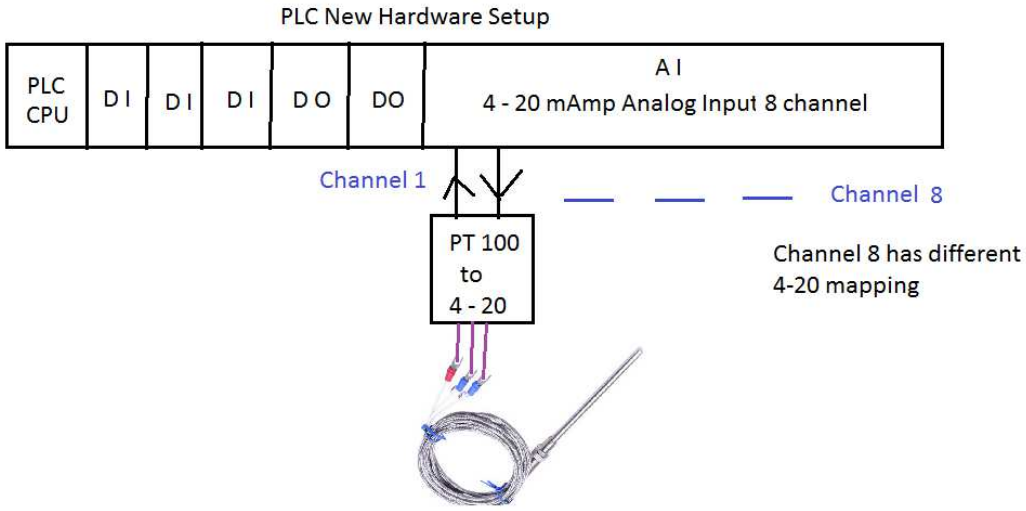
Modbus Driver Implementation For Input Reg Read
Func 0x04 = Read Multiple Input Registers

```
using System;
using System.Text;
using System.IO.Ports;
using System.Threading;
namespace Autoclave
{
    static class ReadAI
    {
        static  private byte _Id { get; set; }
        static  private Int32 _StartAddress { get; set; }
        static  private ushort _NoAI { get; set; }
        static  private SerialPort _sp { get; set; }
        static  public bool success { get; private set;  } // Read only from outside
        static  public string message { get; private set;  }

        static  public short[] GetAI(byte id, Int32 startAdd, ushort NoAI, SerialPort sp)
        {
            try
            {
                _Id = id; _StartAddress = startAdd; _NoAI = NoAI; _sp = sp; success = false; message = "";
                int BytesExpected = 2 * _NoAI + 5;
                byte[] response = new byte[BytesExpected];
                  // A static method can have local variable , response is NOT static

                for (int x = 0; x < (BytesExpected); x++)
                    response[x] = 0;

                _StartAddress = startAdd - 30001;
                _sp.ErrorReceived += new SerialErrorReceivedEventHandler(SpFault);

                if (!_sp.IsOpen) _sp.Open();
                _sp.DiscardInBuffer(); _sp.DiscardOutBuffer();
                byte[] request = MakeRequest();

                // Baud is BITS per sec NOT BYTES per sec.
                int milliSec = (1000 / _sp.BaudRate) * (BytesExpected) * 8 + 100;
              //  100 to device to process the request
                _sp.Write(request, 0, 8);
                // write method blocks. so no need to count for 8 bytes to be sent.
                // device time + 8 bytes to send + retutn bytes
                Thread.Sleep(milliSec);

                int count = 0;
                while (_sp.BytesToRead > 0 || count < BytesExpected) // Or read time out
                {
                    response[count] = ((byte)_sp.ReadByte());
                    count++;
                }
                _sp.Close();
                if (count == 5 && response[1] == 132) // 4 + 128
                {
                    // error response = request code + 0x80 ( 128 )
                    // Protocol error returned from device
                    success = false;
```

```
            message = GetErrorString(response[2]);
            return null;
        }
        else if (!CheckResponse(response))   // check for crc
        {
            success = false;
            message = "ReadAI CRC Failure";
            return null;
        }
        else if (response[1] == 0x4)
        {
            byte[] temp = new byte[2];
            // convert data into int16 and returm it.
            short[] data = new short[_NoAI];
            for (int num = 0; num < _NoAI; num++)
            {
                temp[0] = response[2 * num + 4];      //  Low  Byte
                temp[1] = response[2 * num + 3];      //  High  Byte
                data[num] = BitConverter.ToInt16(temp, 0);
                //  ushort MyNum = (ushort)(( temp[1] << 8) | temp[0] );
            }
            success = true; message = "Success";
            return data;
        }
        else
        {
            success = false; message = "ReadAI Unknown Response";
            return null;
        }
    }
    catch (TimeoutException)
    {
        success = false; message = "ReadAI TimeOut";
        return null;
    }
    catch (Exception ex)
    {
        success = false; message = "ReadAI - " + ex.Message;
        return null;
    }
    finally { _sp.Close(); }

}

static  private bool CheckResponse(byte[] resp)
{
    // crc check calculated at full ///////////
    byte[] bt = new byte[2];
    crcCalculator(resp, ref bt); // calculate crc16 of resp in put it into bt
    if (bt[0] == resp[resp.Length - 2] && bt[1] == resp[resp.Length - 1])
        return true;
    else
        return false; // false = bad
}

static  private byte[] MakeRequest()
{
    // request for reading Analog Input is always 8 bytes long.
    byte[] write = new byte[8];
    write[0] = _Id;    // device id
    write[1] = 0x04;  // Read A I

    byte[] start = BitConverter.GetBytes((ushort)_StartAddress); //  ushort = 2 bytes
    write[2] = start[1]; // msb first
    write[3] = start[0];

    byte[] noReg = BitConverter.GetBytes(_NoAI); //  ushort = 2 bytes
    write[4] = noReg[1]; // msb first
```

```
            write[5] = noReg[0];

            byte[] writeTemp = { write[0], write[1], write[2], write[3], write[4], write[5], 0, 0 };
            // Note two zeros at the end.

            byte[] _crc = new byte[2];
            crcCalculator(writeTemp, ref _crc);

            write[6] = _crc[0];
            write[7] = _crc[1];

            return write;
        }

        static private string GetErrorString(byte n)
        {
            string errorString = "";
            if (n == 1) errorString = "Error Code 1.Illegal Function.";
            else if (n == 2) errorString = "Error Code 2.Illegal Data Address.";
            else if (n == 3) errorString = "Error Code 3.Illegal Data Value.";
            else if (n == 4) errorString = "Error Code 4.Device Failure.";
            else if (n == 5) errorString = "Error Code 5.Device Is Processing Ur Request.Please Wait.";
            else if (n == 6) errorString = "Error Code 6.Device Busy. Ur Request Is Rejected. NAK.";
            else if (n == 7) errorString = "Error Code 7.This Request CanT Be Performed. NAK.";
            else if (n == 8) errorString = "Error Code 8.Memory Pairity Check Failed.";
            else if (n == 10) errorString = "Error Code 10.GateWay Not Available.";
            else if (n == 11) errorString = "Error Code 11.Device Failed To Respond.";
            else errorString = "Custom Error Code.";
            return errorString;
        }

        static private void crcCalculator(byte[] message, ref byte[] CRC)
        {
            ushort CRCFull = 0xFFFF;
            byte CRCHigh = 0xFF, CRCLow = 0xFF;
            char CRCLSB;
            for (int i = 0; i < (message.Length) - 2; i++)
            {
                CRCFull = (ushort)(CRCFull ^ message[i]);

                for (int j = 0; j < 8; j++)
                {
                    CRCLSB = (char)(CRCFull & 0x0001);
                    CRCFull = (ushort)((CRCFull >> 1) & 0x7FFF);

                    if (CRCLSB == 1)
                        CRCFull = (ushort)(CRCFull ^ 0xA001); //  Modbus CRC16 Plolynial 0xA001
                }
            }
            CRC[1] = CRCHigh = (byte)((CRCFull >> 8) & 0xFF);
            CRC[0] = CRCLow = (byte)(CRCFull & 0xFF);
        }

    }
}
```

Application Side Implementation




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Text;
using System.Windows.Forms;

using System.IO.Ports;
using System.Threading;
using System.Diagnostics;

namespace AnalogInputTest
{
    public partial class Form1 : Form
    {
        SerialPort sp = new SerialPort();
        short[] data = new short[8];
        string  AICur = "";
        public Form1() { InitializeComponent();  }
        private void Form1_Load(object sender, EventArgs e)
        {
              AppDomain MyDomain = AppDomain.CurrentDomain;
              MyDomain.UnhandledException += new UnhandledExceptionEventHandler(Handler);

              sp.PortName = "com1";
              sp.BaudRate = 9600;
              sp.DataBits = 8;
              sp.StopBits = StopBits.One;
              sp.Parity = Parity.None;
              sp.ReadTimeout = 1500;
              sp.WriteTimeout = 1500;
              sp.ReceivedBytesThreshold = 1;
              sp.Handshake = Handshake.None; // No software of hardware flow control

              // High Priority to process
              using (Process p = Process.GetCurrentProcess())
                  p.PriorityClass = ProcessPriorityClass.High;

              // High Priority to Thread
              Thread th = new Thread(UpdateTB);
              th.IsBackground = true;
              th.Priority = ThreadPriority.AboveNormal;
              th.Start();
        }
```

```
        void Handler(object sender, UnhandledExceptionEventArgs e)
        {
            Exception exp = (Exception)e.ExceptionObject;
            MessageBox.Show(exp.Message, "Domain Error.", MessageBoxButtons.OK, MessageBoxIcon.Information);
            this.Close();
        }

        private void UpdateTB()
        {
            Control.CheckForIllegalCrossThreadCalls = false;
            while (true) // Loop does not break even during exception
            {
                try
                {
                    AICur = "";
                    // Get Analog Input Address from PLC program
                    // Convert this address to Modbus Equivalent
                    data = ReadAI.GetAI(1, 38305, 8, sp);
                    if (ReadAI.success)
                    {
                        AICur += "Channel 1 = " + data[0].ToString() + "\r\n";
                        AICur += "Channel 2 = " + data[1].ToString() + "\r\n";
                        AICur += "Channel 3 = " + data[2].ToString() + "\r\n";
                        AICur += "Channel 4 = " + data[3].ToString() + "\r\n";
                        AICur += "Channel 5 = " + data[4].ToString() + "\r\n";
                        AICur += "Channel 6 = " + data[5].ToString() + "\r\n";
                        AICur += "Channel 7 = " + data[5].ToString() + "\r\n";
                        textBox_Analog_Input.Text = AICur;

                        label_Status.Text = "";
                    }
                    else
                        label_Status.Text = ReadAI.message;
                }
                catch (Exception ex) { MessageBox.Show(ex.Message, "Error"); }
                finally { Thread.Sleep(1500); }
            }
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (sp.IsOpen)  sp.Close();
        }
    }
}
```