# Introduction

Often in my day I find myself testing and debugging devices that communicate via the modbus protocol. In an effort to save myself time, I have tried several downloadable applications but always came to the same conclusion - "Why pay money for something when it's not that hard to build myself?" Also, given that modbus is a 30 year old standard, it makes it difficult to find current code that is usable in today's popular languages. What follows is a brief description of the madness behind the modbus and a small application that I find myself using daily when testing RS232 modbus devices. The code is hardly robust, but it gets the job done in the sense that it works and it is easy to use and alter - two features I find the most important in a lab setting. I hope you find this of some use!

# Background

Modbus is a serial communications protocol used extensively in the industrial world going all the way back to the 1970s. It is especially useful in situations where one needs to connect to several devices in a network (or slaves) through its implementation of multiple device addresses. But if you found this article, then chances are you already know what modbus is and would rather not be bored to death with the details. For those who want to get their hands dirty with the finer points, here is the place to go for the relevant details.

For the purposes of this article, I'll do a quick rundown of the modbus implementation used and the functions covered.

## Modbus Variants

The modbus protocol comes in two flavors: **RTU** and **ASCII**. RTU is a binary implementation and is often most desirable. As such, this discussion will pertain solely to the RTU standard.

## Modbus Functions

There are numerous functions available in the modbus protocol, yet I haven't found any use for most besides the basic read and write commands. Therefore we will only be looking at implementations of **Function 3 - Read Multiple Registers** and **Function 16 - Write Multiple Registers** commands. Other commands can be learned about by following the link provided in the 'Background' portion of this article.

### Function 3 - Read Multiple Registers Message Framing

#### Request

- **Address** (one byte denoting the slave ID)

- **Function Code** (one byte denoting the function ID, in this case '3')
- **Starting Address** (two bytes representing the modbus register at which to begin reading)
- **Register Quantity** (two bytes denoting the amount of registers to read)
- **CRC** (two bytes containing the cyclical redundancy check checksum for the outgoing message)

#### Response

- **Address** (one byte containing the ID of the slave responding)
- **Function Code** (one byte denoting the function to which the slave is responding, in this case '3')
- **Byte Count** (one byte representing the quantity of bytes being read. Each modbus register is made up of 2 bytes, so this value would be 2 * N, with N being the quantity of registers being read)
- **Register Values** (2 * N bytes representing the values being read)
- **CRC** (two bytes containing the CRC checksum for the incoming message)

### Function 16 - Write Multiple Registers Message Framing

#### Request

- **Address** (one byte denoting the slave ID)
- **Function Code** (one byte denoting the function ID, in this case '16')
- **Starting Address** (two bytes representing the modbus register at which to begin writing)
- **Register Quantity** (two bytes denoting the amount of registers to write)
- **Byte Count** (one byte representing the quantity of bytes being written. Each modbus register is made up of 2 bytes, so this value would be 2 * N, with N being the quantity of registers being written to)
- **Register Values** (2 * N bytes containing the actual bytes being written)
- **CRC** (two bytes containing the cyclical redundancy check checksum for the outgoing message)

#### Response

- **Address** (one byte containing the ID of the slave responding)
- **Function Code** (one byte representing the function being responded to, in this case '16')
- **Starting Address** (two bytes stating the starting register address that was first written to)
- **Register Quantity** (two bytes denoting the quantity of modbus registers that were written to)
- **CRC** (two bytes representing the CRC checksum of the incoming message)

## Error and Exception Coding

The modbus protocol describes numerous exception codes available for each function in its implementation guide. All that is handled in this code is a simple CRC evaluation, as this is hardly a full implementation of the entire modbus protocol. If one chooses to add more error checking, additional procedures can easily be added to the provided code.

# Using the Code

The class *modbus.cs* contains several functions for serial port handling and data transfer. All functions make use of the `SerialPort` class found in the `System.IO.Ports` namespace of .NET 2.0.

One peculiarity of the `SerialPort` class exists in its `DataReceived` event. As has been documented in other articles on this site, the `DataReceived` event, which should fire each time the serial port receives an incoming event, doesn't "actually do this". This prevents truly event driving communications from working properly without the additional fixes and workarounds.

Fortunately, using something like the modbus protocol gives us the benefit of known incoming and outgoing message lengths. This allows us to use the `ReadByte` function as the backbone of any read commands and we can bypass the problem completely. In a perfect world, this would be an event driven class - but in the meantime we'll reject the tradeoff in lost data and tell the port when to read on our own.

## SendFc3 - Read Multiple Registers

Hide Shrink ▲ Copy Code

```
public bool SendFc3(byte address, ushort start,
    ushort registers, ref short[] values)
{
    //Ensure port is open:
    if (sp.IsOpen)
    {
        //Clear in/out buffers:
        sp.DiscardOutBuffer();
        sp.DiscardInBuffer();

        //Function 3 request is always 8 bytes:
        byte[] message = new byte[8];

        //Function 3 response buffer:
        byte[] response = new byte[5 + 2 * registers];

        //Build outgoing modbus message:
        BuildMessage(address, (byte)3, start, registers, ref message);

        //Send modbus message to Serial Port:
        try
        {
```

```
            sp.Write(message, 0, message.Length);

            GetResponse(ref response);
        }
        catch (Exception err)
        {
            modbusStatus = "Error in read event: " + err.Message;
            return false;
        }

        //Evaluate message:
        if (CheckResponse(response))
        {
            //Return requested register values:
            for (int i = 0; i < (response.Length - 5) / 2; i++)
            {
                values[i] = response[2 * i + 3];
                values[i] <<= 8;
                values[i] += response[2 * i + 4];
            }

            modbusStatus = "Read successful";
            return true;
        }
        else
        {
            modbusStatus = "CRC error";
            return false;
        }
    }
    else
    {
        modbusStatus = "Serial port not open";
        return false;
    }
}
```

This public function accepts four variables - `address` (slave ID), `start` (register to begin read at), `registers` (quantity of registers to read) and `values` (reference to a byte array to contain the read values). The function will return `true` for a successful read or `false` for an erroneous read.

The `BuildMessage()` function simply places each byte into the desired modbus message format as well as passes this message to the CRC calculator. `GetResponse()` reads a byte stream of set length from the serial port and places this result into the `response[]` array. These functions are very straightforward and can be examined by downloading the sample code. Following the read and write routines, the requested data is handled by placing it into the `short[]` array passed by reference to this function. These values are then made available to the user application.

The public string `modbusStatus` contains pertinent information at various points during the modbus communication and makes this information available to the user application during program execution. This can be seen throughout the sample application.

## SendFc16 - Write Multiple Registers

Hide Shrink ▲ Copy Code

```
public bool SendFc16(byte address, ushort start,
    ushort registers, short[] values)
{
    //Ensure port is open:
    if (sp.IsOpen)
    {
        //Clear in/out buffers:
        sp.DiscardOutBuffer();
        sp.DiscardInBuffer();

        //Message is 1 addr + 1 fcn + 2 start + 2 reg + 1 count +
            2 * reg vals + 2 CRC
        byte[] message = new byte[9 + 2 * registers];

        //Function 16 response is fixed at 8 bytes
        byte[] response = new byte[8];

        //Add bytecount to message:
        message[6] = (byte)(registers * 2);

        //Put write values into message prior to sending:
        for (int i = 0; i < registers; i++)
        {
            message[7 + 2 * i] = (byte)(values[i] >> 8);
            message[8 + 2 * i] = (byte)(values[i]);
        }

        //Build outgoing message:
        BuildMessage(address, (byte)16, start, registers, ref message);

        //Send Modbus message to Serial Port:
        try
        {
            sp.Write(message, 0, message.Length);
            GetResponse(ref response);
        }
        catch (Exception err)
        {
            modbusStatus = "Error in write event: " + err.Message;
            return false;
        }

        //Evaluate message:
        if (CheckResponse(response))
        {
            modbusStatus = "Write successful";
```

```
            return true;
        }
        else
        {
            modbusStatus = "CRC error";
            return false;
        }
    }
    else
    {
        modbusStatus = "Serial port not open";
        return false;
    }
}
```

This function behaves in a similar fashion to `SendFc3`. Again we are passing four variables to the function, only this time instead of passing a byte array by reference to receive the read values, we are passing a populated array of values to use when writing the relevant registers. Again, we receive a `boolean` value depicting the success, or lack thereof, of the write command.

As previously mentioned, the `CheckResponse` function does nothing besides a simple CRC check. The CRC is recalculated for the incoming message and this value is compared against that passed as part of the message itself - hence checking the validity of the response. Any message framing error should result in a CRC exception.

# Points of Interest

Again, it's important to reiterate the downfalls of the event handling in the `SerialPort DataReceived`event. A simple Google search will locate several frustrated coders explaining identical situations with few solutions suggested. This site contains one such solution and whether or not to pursue this line of coding is up to the individual. A better implementation of the modbus protocol would contain an override for this event and a more finely tuned data reception function.

Note that the public functions found in this `modbus` class were built with the intention that they can be used in dedicated threads in a user-made application. The sample application demonstrates the use of these functions in a thread created by the `System.Timers.Timer.Elapsed` event. It should be easy to find other uses that will fit your specific needs.