

# The Basics of Modbus Monitoring with PRTG Network Monitor (With Custom Sensor Script)

Article explains how Modbus/TCP works and shows how to monitor Modbus/TCP enabled devices with PRTG Network Monitor



The [Modbus protocol](#) has been around forever (since 1979 to be exact) and is used by many industrial systems, but also energy systems like [heat pumps](#) and [solar converters](#). Initially it was used via serial communications, then – in 2007 – a TCP based version was created that communicates via TCP, usually using port 502.

The following article explains how Modbus/TCP works and shows how I can monitor Modbus/TCP enabled devices with [PRTG Network Monitor](#) using a simple PowerShell script.

I wish I could share a one-size-fits-all script that just works for most people. But unfortunately this task usually requires some hacking and coding...

## What makes Modbus special

The Modbus protocol allows a client to request numeric values from a device using a TCP request. The data is organized in numbered “registers” and the data format is almost crude.

Compared to many other (more sophisticated) communication standards the Modbus/TCP standard stands out for several reasons:

- It is simple and has an almost microscopic bandwidth usage: a usual data request only consists of two handful of bytes sent and about two or three handful of bytes received. Here is a sample communication trace:

```
[TCP]>Tx > 20:50:34:899 - 00 03 00 00 00 06 03 04 77 41 00 02  
[TCP]>Rx > 20:50:35:098 - 00 03 00 00 00 07 03 04 04 00 C1 60 22
```

- It has no authentication: If I can access the TCP port 502 of a device over the network I can read (and write!) device data using Modbus which does make it a potential security challenge
- It has no encryption
- The standard is quite lax: The vendors can define the format of the returned data as they wish, so it is the receiver’s problem to adapt to the data format.

The last fact makes it hard for monitoring systems to digest the data in a standardized way. The original protocol only allows 16-bit integers. But many vendors encode the data as word, or int16, or uint16, or uint32, or unit64, or float, or string, etc., often mixing the formats as it fits the data. Some vendors send the high byte first, other the low byte. Some vendors send the high word first, other the low word.

Sometimes the vendor documentations are not very exact and it can require some “hacking” and guessing to find out what that stream of bytes actually means. And after successfully interpreting the byte code it may still be necessary to finally process the data further. E.g. sometimes a value of 37.2 is transferred as 372 and you are expected to divide that by 10 before using it.

## Interpreting Modbus data can be a mess

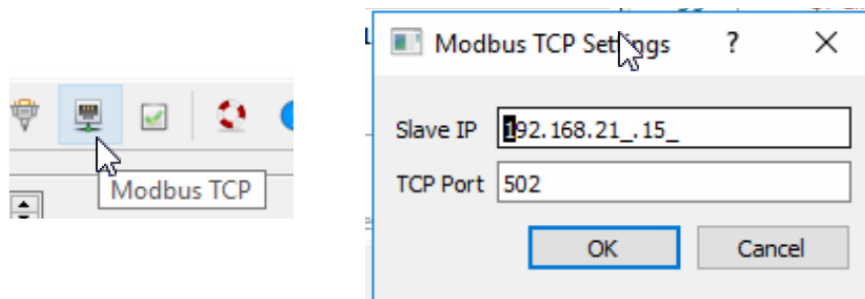
This mess makes it necessary for the receiver of the data to surrender to the vendors' will. In PRTG this means that most of the time I have to write a specific script for each device so I can translate the data for PRTG individually for each device(-type).

I need the following things to set up Modbus monitoring:

- The documentation from the vendor
- Access to the device (for hacking/experimentation)
- An interactive Modbus software
- A script editor and a demo script to start with
- And some time

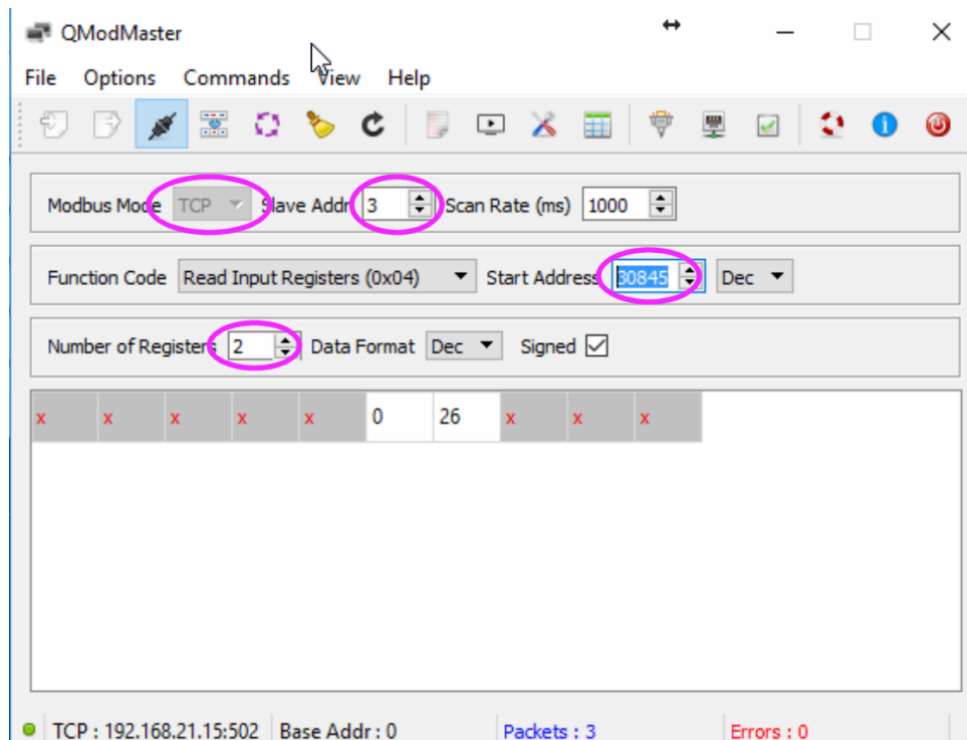
## Let's get started

I started with the free QModMaster software ([SourceForge download](#)). First I clicked on the **Ethernet icon** and entered the **Slave IP** address of my device (502 is the standard port for Modbus):



Then I ...

- selected the **Modbus Mode** "TCP"
- entered the **Slave Address** (you get this from the device, some devices even ignore it)
- Selected the **Function Code** as "Read Input Registers (0x04)"
- Entered a value for **Start Address** (from vendor documentation)
- Selected the **Number of Registers** I want to read (again from vendor documentation)

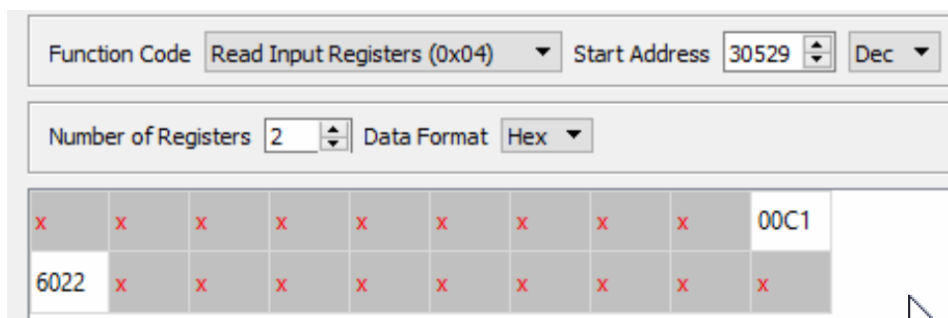


In this screenshot I read the battery charge level from my solar battery, which is 26%. OK, that wasn't actually so hard.

Short Note: Some vendors start numbering their registers at 1, others start with number 40001 (which was the initial start address in the protocol). One vendor showed the number 1 in the documentation which I had to convert into 40001+1 to actually make it work.

## Interpreting the data

The next register I wanted to access was a counter for the total energy produced by my solar panels. This value is stored as 32-bit integer in the two 16-bit registers #30529 and #30530.



I got \$00c1 and \$6022 which as int32 translates to 12.673.058 or 12.673 MWh, which I could compare to the data by the solar system itself.



Yes, this work is ugly... I did this for all the metrics I wanted to get from the device the I was ready to...

## Write a monitoring script for PRTG

To get the data into PRTG I am using an [EXE/Script Advanced sensor](#) in [PRTG](#). The script was written in Powershell.

In the Windows Powershell ISE I can edit and test the script:

```

1 # Modbus Monitoring Sample Script for PRTG
2 # via Modbus Protocol
3
4 param(
5     [int]$port = 502 # Standard Modbus Port
6 )
7
8 function modbusread ([string]$remoteHost, [int]$port, [int]$startaddress, [int]$bytecount, [string]$unitID)
9 {
10     write-host "Reading Register # $startaddress"
11
12     # Build Request Data
13
14     [byte[]]$sendbuffer=00,110 # Transaction Identifier
15     $sendbuffer+=00,00 # Protocol identifier
16     $sendbuffer+=00,06 # Length
17     $sendbuffer+=03 # Unit ID
18     $sendbuffer+=04 # Function Read Input Registers
19     $sendbuffer+=([byte]::Truncate(($startaddress)/256)),([system.byte]($startaddress)%256)
20     $sendbuffer+=00,($bytecount)
21
22     # Send Request Data
23
24     $tcpclient = new-object System.Net.Sockets.TcpClient($remoteHost, $port)
25     $netStream = $tcpclient.GetStream()
26     $netStream.write($sendbuffer,0,$sendbuffer.length)
27     start-sleep -milliseconds 50
28
29     # Receive Data
30
31     [byte[]]$recbuffer = new-object System.Byte[] ([int]($bytecount+9))
32
33     $netStream.read($recbuffer,0,$recbuffer.length)
34
35     # Parse Response
36     $transactionID = [int]($recbuffer[0])
37     $protocolID = [int]($recbuffer[1])
38     $length = [int]($recbuffer[2])
39     $unitID = [int]($recbuffer[3])
40     $functionCode = [int]($recbuffer[4])
41     $startAddress = [int]($recbuffer[5]*256+$recbuffer[6])
42     $byteCount = [int]($recbuffer[7])
43
44     # Output Data
45     write-host "==== Power from PV ======"
46     write-host "Reading Register #30775"
47     write-host "Received int= -2147483648"
48     write-host "==== Solar Energy Total ======"
49     write-host "Reading Register #30529"
50     write-host "Received uint= 12673058"
51 }
52
53 modbusread "192.168.1.100" 502 30775 4 1
54 modbusread "192.168.1.100" 502 30529 4 1
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Let's go...

```

==== Power from PV =====
Reading Register #30775
Received int= -2147483648
==== Solar Energy Total =====
Reading Register #30529
Received uint= 12673058

```

Please look at the following aspects when you edit the script for your needs:

- You may need to set the “Unit ID” (line 17 of the script, this is set in the device’s settings)

- You may need to adapt the data processing and conversion (modbusread() function)
- You must set the IP address
- For each metric you must add and edit one of the lines with the function calls to onedataset()

## Adding Metrics to the script

You can add/rename metrics in these code lines:

```
$remoteHost = "192.168.21.14"

$prtgresult+=onedataset "Power from PV" 30775 "W" 1 "Absolute" "signed"
$prtgresult+=onedataset "Solar Energy Total" 30529 "Wh" 1 "Absolute"
$prtgresult+=onedataset "Solar Energy Today" 30535 "Wh" 1 "Absolute"
```

The parameters of the onedataset function() are:

- name: The name of the metric (which is shown in PRTG's UI)
- theid: the number of the modbus register
- unit: The unit string (which is shown in PRTG's UI, e.g. °C, W, Byte)
- divider: A number the metric needs to be divided by
- type: This can be "Absolute" (for most metrics, uses the value as it is) or "Difference" (for counters, PRTG calculates the difference to the previous sensor value and displays it on a "per time" basis, e.g. bytes/sec, see [PRTG's documentation](#))
- signed: Must be "signed" if the metric is a signed int32, otherwise the registers' data will be interpreted as unsigned int32.

When I run this script in the IDE I get:

```
Let's go...
==== Power from PV ====
Reading Register #30775
Received int= -260
==== Solar Energy Total ====
Reading Register #30529
```

```
Received uint= 12673058
==== Solar Energy Today ====
Reading Register #30535
Received uint= 6896
```


And the output sent to PRTG a XML is:

```
<?xml version="1.0" encoding="Windows-1252" ?>
<prtg>
  <result>
    <channel>Power from PV</channel>
    <customunit>W</customunit>
    <value>-260</value>
    <float>1</float>
    <mode>Absolute</mode><SpeedTime>Hour</SpeedTime>
  </result>
  <result>
    <channel>Solar Energy Total</channel>
    <customunit>Wh</customunit>
    <value>12673058</value>
    <float>1</float>
    <mode>Absolute</mode><SpeedTime>Hour</SpeedTime>
  </result>
  <result>
    <channel>Solar Energy Today</channel>
    <customunit>Wh</customunit>
    <value>6896</value>
    <float>1</float>
    <mode>Absolute</mode><SpeedTime>Hour</SpeedTime>
  </result>
</prtg>
```

Finally I needed to copy the script file **Modbus Sample Script.ps1** into the folder

**C:\Program Files\PRTG Network Monitor\Custom Sensors\EXEXML**

and I set the [Execution Policy properly](#). In PRTG's UI I set up a EXE/Script Advanced Sensor



EXE/Script ⓘ Modbus Sample Script.ps1

and I am finished.



Finally, here is my script, feel free to use and adapt it for your needs:

```
# Modbus Monitoring Sample Script for PRTG
# via Modbus Protocol

param(
    [int]$port = 502    # Standard Modbus Port
)

function modbusread ([string]$remoteHost, [int]$port, [int]$startaddress, [int]$bytecount, [string]$forma) {

    write-host "Reading Register #$startaddress"

    # Build Request Data

    [byte[]]$sendbuffer=00,110 # Transaction Identifier
    $sendbuffer+=00,00         #Protocol identifier
    $sendbuffer+=00,06         #Length
    $sendbuffer+=03            #Unit ID
    $sendbuffer+=04            #Function Read Input Registers
    $sendbuffer+=([byte]([math]::Truncate(($startaddress)/256)),([system.byte]((($startaddress)%256)))
    $sendbuffer+=00,($bytecount)

    # Send Request Data

    $tcpclient = new-object System.Net.Sockets.TcpClient($remoteHost, $port)
    $netStream = $tcpclient.GetStream()
    $netStream.write($sendbuffer,0,$sendbuffer.length)
    start-sleep -milliseconds 50

    # Receive Data

    [byte[]]$recbuffer = new-object System.Byte[] ([int]($bytecount+9))
    $receivedbytes = $netStream.Read($recbuffer, 0, [int]($bytecount+9));
    $netStream.Close()
    $tcpclient.Close()

    # Process Data

    $resultdata = $recbuffer[9..($recbuffer[8]+8)]

    # depending on the data from the specific device you may need to reverse the byte order

    [byte[]] $bytes = $resultdata[3],$resultdata[2],$resultdata[1],$resultdata[0] # need to reverse byte order

    if ($signed -eq "signed") {
        $result=[bitconverter]::ToInt32($bytes,0);
        write-host "Received int=" $result
    }
    else {

```



```

        $result=[bitconverter]::ToUInt32($bytes,0);
        write-host "Received uint=" $result
    }
    $result
}

# Process One Dataset

function onedataset([string]$name, [int]$theid, [string]$unit, [int]$divider, [string]$type, [string]$signed) {
    write-host "==== $name ===="
    $value=((modbusread $remoteHost $port ($theid) 4 $signed)/$divider)
    if ($value -eq 4294967295 -Or $value -eq 2147483648 -Or $value -eq -2147483648) # sometimes these values mean "not available"
    { $value=0 }

    "    <result>'r'n"
    "    <channel>"+$name+"</channel>'r'n"
    "    <customunit>"+$unit+"</customunit>'r'n"
    "    <value>"+($value/$divider)+"</value>'r'n"
    "    <float>1</float>'r'n"
    "    <mode>"+$type+"</mode><SpeedTime>Hour</SpeedTime>'r'n"
    "    </result>'r'n"
}

# Main code

write-host "Let's go..."
[string]$prtgresult=""
$prtgresult+="xml version='1.0' encoding='Windows-1252' ?&gt;'r'n"
$prtgresult+="<prtg&gt;'r'n"
[bool]$errorfound = $false

try {

    $remoteHost = "192.168.21.14"

    $prtgresult+=onedataset "Power from PV" 30775 "W" 1 "Absolute" "signed"
    $prtgresult+=onedataset "Solar Energy Total" 30529 "Wh" 1 "Absolute"
    $prtgresult+=onedataset "Solar Energy Today" 30535 "Wh" 1 "Absolute"

    $prtgresult+="<&lt;/prtg&gt;"
}
catch {
    write-host "Unable to Connect and retrieve data $_.Exception.Message"
    $prtgresult+="<error&gt;2&lt;/error&gt;'r'n"
    $prtgresult+="<text&gt;Unable to Connect and retrieve data " +($_.Exception.Message + " at " + $_.InvocationInfo.PositionM
    $prtgresult+="<&lt;/prtg&gt;"
    $errorfound = $true
}

if ($errorfound) {
    write-host "Error Found. Ending with EXIT Code" ($prtgresult).prtg.error
}
write-host "Sending PRTGRESULT to STDOUT"
$prtgresult
</pre

```

## Links

I found the following links helpful for this work:

- QModMaster Software: <https://sourceforge.net/projects/qmodmaster/>
- Modbus for Field Technicians:  
[http://www.modbusbacnet.com/includes/pdf/MODBUS\\_2010Nov12.pdf](http://www.modbusbacnet.com/includes/pdf/MODBUS_2010Nov12.pdf)
- “Simply Modbus” documentation: <http://www.simplymodbus.ca/TCP.htm>
- Pluggit Monitoring for PRTG (my script is based on this script):  
<https://www.msxfaq.de/tools/prtg/prtg-pluggit.htm>

Share this:



**Author: Dirk Paessler**

Founder and Chairman, Paessler AG [View all posts by Dirk Paessler](#)

February 4, 2019 / Smart Home and IoT /

---

## 7 thoughts on “The Basics of Modbus Monitoring with PRTG Network Monitor (With Custom Sensor Script)”

---



**Chris M.**

February 4, 2019 at 9:30 am

Great! Nice article, thanks for sharing your knowledge and the scripts 😊  
I'm pretty sure that a lot of people benefit from it.

★ Liked by you

---



**kgulle**

February 4, 2019 at 1:02 pm

great work done.  
Thank you.

★ Liked by you

---



**Alex**

May 6, 2019 at 7:11 am

There may be a small bug in the onedataset function? Line 59 and line 66 both have “/\$divider”, which means that values going into PRTG are out by \$divider. Simply removing the one on line 66 worked for me.

Anyway, thanks a million for writing such a helpful post—thanks to your help, I’m now able to monitor our <https://www.victronenergy.com/panel-systems-remote-monitoring/venus-gx> in PRTG!

My final code was:

```
# Modbus Monitoring Sample Script for PRTG
```

```
param(  
[int]$port = 502 # Standard Modbus Port  
)
```

```
function modbusread ([string]$remoteHost, [int]$port, [int]$startaddress,  
[int]$bytecount,[string]$forma) {
```

```
write-host “Reading Register # $startaddress”
```

```
# Build Request Data
```

```
[byte[]]$sendbuffer=00,110 # Transaction Identifier  
$sendbuffer+=00,00 # Protocol identifier  
$sendbuffer+=00,06 # Length  
$sendbuffer+=100 # Unit ID  
$sendbuffer+=04 # Function Read Input Registers  
$sendbuffer+=[byte]([math]::Truncate(($startaddress)/256)),([system.byte]  
(($startaddress)%256))  
$sendbuffer+=00,($bytecount)
```

```
# Send Request Data
```

```
$tcpclient = new-object System.Net.Sockets.TcpClient($remoteHost, $port)  
$netStream = $tcpclient.GetStream()
```

```
$netStream.write($sendbuffer,0,$sendbuffer.length)
```

```
start-sleep -milliseconds 50
```

```
# Receive Data
```

```
[byte[]]$recbuffer = new-object System.Byte[] ([int]($bytecount+9))
```

```
$receivedbytes = $netStream.Read($recbuffer, 0, [int]($bytecount+9));
```

```
$netStream.Close()
```

```
$tcpclient.Close()
```

```
# Process Data
```

```
$resultdata = $recbuffer[9..($recbuffer[8]+8)]
```

```
# depending on the data from the specific device you may need to reverse the byte  
order
```

```
# [byte[]] $bytes = $resultdata[3],$resultdata[2],$resultdata[1],$resultdata[0] # int32
```

```
[byte[]] $bytes = $resultdata[1],$resultdata[0] # int16
```

```
if ($signed -eq "signed") {
```

```
# $result=[bitconverter]::ToInt32($bytes,0);
```

```
$result=[bitconverter]::ToInt16($bytes,0);
```

```
write-host "Received int=" $result
```

```
}
```

```
else {
```

```
# $result=[bitconverter]::ToUInt32($bytes,0);
```

```
$result=[bitconverter]::ToUInt16($bytes,0);
```

```
write-host "Received uint=" $result
```

```
}
```

```
$result
```

```
}
```

```
# Process One Dataset
```

```
function onedataset([string]$name, [int]$theid, [string]$unit, [int]$divider,
[string]$type, [string]$signed) {
write-host "==== $name ===="
$value=((modbusread $remoteHost $port ($theid) 2 $signed)/$divider) # 2 for int16
or 4 for int32
if ($value -eq 4294967295 -Or $value -eq 2147483648 -Or $value -eq -2147483648) #
sometimes these values mean "not available"
{ $value=0 }
```

```
" `r`n"
" "+$name+"`r`n"
" "+$unit+"`r`n"
" "+$value+"`r`n"
" 1`r`n"
" "+$type+"Hour`r`n"
" `r`n"
}
```

# Main code

```
write-host "Let's go..."
[string]$prtgresult=""
$prtgresult+="`r`n"
$prtgresult+="`r`n"
[bool]$errorfound = $false
```

```
try {
$remoteHost = "192.168.21.14"
```

```
$prtgresult+=onedataset "Battery Voltage (System)" 840 "V DC" 10 "Absolute"
$prtgresult+=onedataset "Battery Current (System)" 841 "A DC" 10 "Absolute"
"signed"
$prtgresult+=onedataset "Battery Power (System)" 842 "W" 1 "Absolute" "signed"
```

```
$prtgresult+="`r`n"
}
catch {
```

```
write-host "Unable to Connect and retrieve data $_.Exception.Message"
$prtgresult+=" 2`r`n"
$prtgresult+=" Unable to Connect and retrieve data "+($_.Exception.Message +" at "+
$_InvocationInfo.PositionMessage)+"`r`n"
$prtgresult+="""
$errorfound = $true
}

if ($errorfound) {
write-host "Error Found. Ending with EXIT Code" ($prtgresult).prtg.error
}
write-host "Sending PRTGRESULT to STDOUT"
$prtgresult
```

★ Liked by you

---



**Dirk Paessler** 👤

May 6, 2019 at 11:14 am

Thanks, Alex, for your comment.

★ Liked by [1 person](#)

---



**Benjamin Day**

June 7, 2019 at 11:51 pm

Hey Dirk! Glad to see you're still getting some time to code! I had a user use this, and it worked fine, but I think it tried to send the whole output to PRTG. Should it just be sending PRTG the XML?

★ Like

---



**Dirk Paessler** 👤

June 11, 2019 at 9:35 pm

Hi Ben, the debug code sent with “write-host” ist not sent to PRTG. If you get error messages like

XML: The returned XML does not match the expected schema. (code: PE233) (code: PE231)

then make sure the execution policy for 16-bit (!!!) PowerShell Scripts is set correctly, as described here:

<https://dirkpaessler.blog/2019/01/28/pe233-and-pe231-error-with-a-powershell-script-for-a-custom-xml-sensor-for-prtg/>

 Like



**ralliredgts**

June 7, 2019 at 11:52 pm

Hey Dirk! Glad to see you're still getting some time to code! I had a user use this, and it worked fine, but I think it tried to send the whole output to PRTG. Should it just be sending PRTG the XML?

 Like