Guideline for Modbus

Modbus is a communication protocol and is considered a quasi-standard in automation technology. It is based on binary data notation and uses a master-slave mechanism (client-server architecture). The principle of data transmission applied for this is simple: Address, instruction code, length specification, data content, and checksum are forming make up the packets. This principle can be used universally and explains why Modbus is very widely used.

But why do discrepancies come up in relation with Modbus communication time and again?

The beginning is the end

No, we are not kidding. It is the most important fact to consider: the order of the data bits and bytes. We are talking about *"endianness"* or *"byte order"*.

Precisely, because it is such a simple and universal protocol, using Modbus requires dealing with many different computer and memory architectures. This is where endianness comes into play.

Modbus uses 16-bit registers for data exchange. So the smallest unit has a size of 16 bits or, correspondingly, 2 bytes in this case. Accordingly, 8-bit architectures are using two memory units for each value. At this point, the question of sequence is raised for the first time.

When two parts of a value are combined to obtain a total value, like combining two 8-bit values to one 16-bit value in this case, you have to start thinking about their order. For example, let us take a value of 4660 (or 0x1234 in hexadecimal terms). This 16-bit value consists of two 8-bit parts 0x12 and 0x34. But, they may be stored in the memory like this:

Memory address	Big-Endian	Little-Endian
1	0x12	0x34
2	0x34	0x12

In this case, **big-endian** indicates that the part with the most significant value is at the lowest memory address which means that the presentation is following the common and classical Arabic numeral notation. The digits written first (e.g. thousands) have a higher significance than those at the (e.g. unit digits).

Little-endian means that the least-significant part is found at the lowest memory address. This contradicts classical Arabic numerical notation, and rather follows the sequence of processing which is used in computation, i.e. you start at the unit position just like column addition. This can bring advantages in arithmetic terms, and is used for this reason in every x86-compatible computer architecture, i.e. also in your PC.

Date and time of day are often mentioned as an analogy. In German-speaking countries, the little-endian, i.e. the least significant value, appears first when indicating a date: 01.02.2003. But the time of day is big-endian: 12:34.

This is relevant for the Modbus since serial communication with an 8-bit data width is used almost in every case. Even for Modbus TCP, data is arranged just like in serial communication.

Supposedly, it is to ensure readability in data streams, or also because of the architecture of earlier PLC, that Modbus applies a big-endian notation. This means that the byte with the highest significance is transmitted first: First comes 0x12 and then 0x34. When Modbus encounters a typical x86 architecture on a PC, a conflict arises as this architecture has 0x34 at the first position in the memory or wants to write it first to the memory. So this must be turned around or swapped.

In case of 16-bit values, you must consider or swap the order of single bytes. When you go to a higher value range like 32-bit values or even 64-bit-values, this variability becomes exponential. But you will not find every combination as architectures will normally remain faithful to their endianness.

Let us mention a specific example: Take the number of 305419896 or 0x12345678 in hexadecimal notation. In this case, the byte order may look like this:

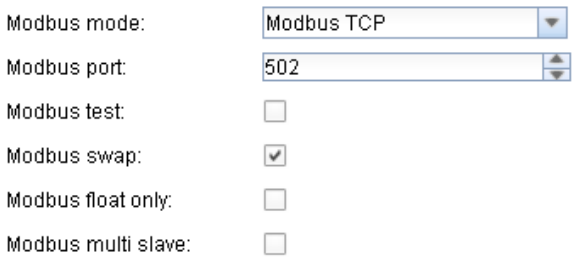
Order	Modbus	Big-Endian*	Big-Mix*	Little-Mix*	Little-Endian
1st	0x12	0x12	0x34	0x56	0x78
2nd	0x34	0x34	0x12	0x78	0x56
3rd	0x56	0x56	0x78	0x12	0x34
4th	0x78	0x78	0x56	0x34	0x12

*Fancy names

So it can quickly happen that a software on a PLC or on a PC does not transmit a number as 0x12, 0x34, 0x56, 0x78 in Modbus but rather as 0x78, 0x56, 0x34, 0x12 (2018915346 decimally), or even in one of the mixed forms if proper care is not taken or for lack of true competence. To deal with this, it is good when you have the possibility to swap bytes, at least to switch from big to little endianess.

In our modern Modbus devices, you can change swapping from big-endian to little-endian. The

"Modbus swap" switch is used for doing this. When it is activated, little-endian notation will be applied.



As the saying goes: This was the first trick. The second one will follow right away ... The second part of our Modbus guideline will show how to properly address registers or storage positions.

In the first part of this blog post about the Modbus communications protocol, we have had a look at endianness, i.e., at the sequence of data representation. Now we would like to describe the proper addressing of registers.

Everything to 0?

Another pitfall to avoid in Modbus is to address registers or, respectively, storage locations in a wrong way. These have already appeared in the tables above: 1st, 2nd, 3rd, 4th. The first value is written to line 1 or to address 1. But this would be wrong when speaking "Modbusian" terms. As it is a binary protocol, logical counting starts at zero in the Modbus world. Accordingly, the first value is written to the register at address 0.

So, when you want to query the first value or, correspondingly, the first register, you have to use address 0. This is easy to do on a binary level. But now the human factor comes into play as the first value would be number 1 to our minds. For this reason, many Modbus components use a human way of addressing, and start counting at 1.

To make things even more confusing, this 1 will, nonetheless, automatically turn into a 0 at the Modbus interface and the counterpart will then respond with a 0, too.

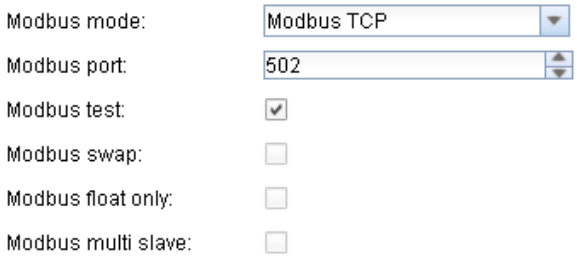
To avoid this problem, the same way of counting must be used on both sides.

In our devices, we follow Modbus, and use a counter which starts at 0. Accordingly, the first 10 registers have addresses 0 to 9, for example. We do not allow any shift by +1.

If the remote station starts counting at 1, you will then have to increase every address by 1 contrary to the notation used in our devices.

How can you find out about this?

To specifically take care of these two aspects, we have integrated a so-called test mode in our devices:



This test mode activates a static response data record which can be used for testing a remote station. Endianness and counting method will not coincide unless the numbers exactly correspond to the specifications as described in our manual.

Address	Value	Description	Decoded value
0	0xD080	Serial number of the device, upper word	0xD0800DC1: last digits of the MAC address: 68:91:D0:80:0D:C1
1	0x0DC1	Serial number of the device, lower word	
2	0x0002	Version of the communication protocol of the device	2
3	0x0084	Software version of the device	0x84 = 132: Version 1.32
4	0x5CE5	System time of the device (timestamp), upper word	0x5CE55EAC = 1559054252: Wednesday, 22 May 2019, 16:37:32 GMT+2
5	0x5EAC	System time of the device (timestamp), lower word	
6	0x0000	Blank register	
7	0x0100	Type field of the data set in the upper byte	0x01: type is device entry
8	0x0000	Blank register	
9	0x0000	Blank register	
10	0x00BC	Serial number of the meter, upper word	0xBC614E = 12345678
11	0x614E	Serial number of the meter, lower word	
12	0x0443	Manufacturer code of the meter (see Section 10.3)	0x0443: ABC
13	0x0102	Version (upper byte) and medium (lower byte) of the meter	0x01: version = 1, 0x02: medium = 2 (Electricity)
14	0x5CE5	Readout time of the meter (timestamp), upper word	0x5CE55EAC = 1559054252: Wednesday, 22 May 2019, 16:37:32 GMT+2
15	0x5EAC	Readout time of the meter (timestamp), lower word	
16	0x0000	Blank register	
17	0x0200	Type field of the data set in the upper byte	0x02: type is meter entry
18	0x0000	Flags in the lower byte	0x00: Meter correctly read and all values up to date
19	0x0000	Blank register	
20	0x0000	Meter value (integer), highest word	0xBC614E = 12345678: Resulting meter value: 12345678 * 10^{-4} = 1234.5678 Wh
21	0x0000	Meter value (integer)	
22	0x00BC	Meter value (integer)	
23	0x614E	Meter value (integer), lowest word	
24	0x449A	Meter value (floating point), upper word	0x449A522B = 1234.567749234375 (rounding error using *FLOAT32*)
25	0x522B	Meter value (floating point), lower word	
26	0xFFFC	Scaling factor (*n*th power of 10)	0xFFFC = -4: factor = 10^{-4}
27	0x0005	Type field of the data set in the upper byte and unit in the lower byte (see Table 25)	0x00: type is meter value entry 0x05: unit = Wh
28	0x5CE5	Provided time of the meter value (timestamp), upper word	0x5CE55EAC = 1559054252: Wednesday, 22 May 2019, 16:37:32 GMT+2
29	0x5EAC	Provided time of the meter value (timestamp), lower word	

If your remote station supports Float32 as a data format, we always recommend to refer to registers 24 and 25 for testing.

If the number at your remote station does not correspond exactly to 1234.5677490234375 (yes, minor deviations may occur in Float32) something is wrong. If you see a value of 23781078392032251084 8, for instance, you would be reading out registers 23 and 24, which would indicate a counting method starting at 1.

To check Float32 values you can use the following page among others: https://www.h-schmidt.net/FloatConverter/IEEE754.html