The Roboteq® Modbus Implementation User Manual

V1.1, December 21, 2017

Visit www.roboteq.com to download the latest revision of this manual

Table of Contents

1. Introduction

The Roboteq® Modbus Implementation User manual contains information about how Roboteq implemented Modbus protocol in controllers.

1.1 What is Modbus

Modbus is a serial communication protocol developed by Modicon published by Modicon® in 1979 for use with its programmable logic controllers (PLCs). In simple terms, it is a method used for transmitting information over serial lines between electronic devices. The device requesting the information is called the Modbus Master and the devices supplying information are Modbus Slaves. In a standard Modbus network, there is one Master and up to 247 Slaves, each with a unique Slave Address from 1 to 247. The Master can also write information to the Slaves.

The official Modbus specification can be found at www.modbus.org/specs.php.

1.2 Modbus object types

The following is a table of object types provided by a Modbus slave device to a Modbus master device:

Object type	Access	Size
Coil	Read/Write	1 bit
Discrete Input	Read Only	1 bit
Input Register	Read Only	16 bits
Holding Register	Read/Write	16 bits

1.3 Protocol versions

Versions of the Modbus protocol exist for serial port and for Ethernet and other protocols that support the Internet protocol suite. There are many variants of Modbus protocols:

- **Modbus RTU:** This is used in serial communication and makes use of a compact, binary representation of the data for protocol communication. The RTU format follows the commands/data with a cyclic redundancy check checksum as an error check mechanism to ensure the reliability of data. Modbus RTU is the most common implementation available for Modbus. A Modbus RTU message must be transmitted continuously without inter-character hesitations. Modbus messages are framed (separated) by idle (silent) periods.

- **Modbus ASCII:** This is used in serial communication and makes use of ASCII characters for protocol communication. The ASCII format uses a longitudinal redundancy check checksum. Modbus ASCII messages are framed by leading colon (":") and trailing newline (CR/LF).

- **Modbus TCP/IP or Modbus TCP:** This is a Modbus variant used for communications over TCP/IP networks, connecting over port 502. It does not require a checksum calculation, as lower layers already provide checksum protection.

- **Modbus over TCP/IP or Modbus over TCP or Modbus RTU/IP:** This is a Modbus variant that differs from Modbus TCP in that a checksum is included in the payload as with Modbus RTU.

Data model and function calls are identical for the previous 4 variants of protocols; only the encapsulation is different.

1.4 Communication and devices

Each device intended to communicate using Modbus is given a unique address. On Ethernet, any device can send out a Modbus command, although usually only one master device does so. A Modbus command contains the Modbus address of the device it is intended for (1 to 247). Only the intended device will act on the command, even though other devices might receive it (an exception is specific broadcast commands sent to node 0, which are acted on but not acknowledged). All Modbus commands contain checksum information, to allow the recipient to detect transmission errors. The basic Modbus commands can instruct an RTU to change the value in one of its registers, control or read an I/O port, and command the device to send back one or more values contained in its registers.

There are many modems and gateways that support Modbus, as it is a very simple protocol and often copied. Some of them were specifically designed for this protocol. Different implementations use wireline, wireless communication, such as in the ISM band, and even Short Message Service (SMS) or General Packet Radio Service (GPRS). One of the more common designs of wireless networks makes use of mesh networking. Typical problems that designers have to overcome include high latency and timing issues.

1.5 Frame format

A Modbus frame is composed of an Application Data Unit (ADU), which encloses a Protocol Data Unit (PDU):

- ADU = Address + PDU + Error check,
- PDU = Function code + Data.

Note:
The byte order for values in Modbus data frames is big-endian (MSB, most significant byte of a value received first).

All Modbus variants choose one of the following frame formats:

1.5.1 Modbus RTU frame format

Name	Length (bytes)	Description
Address	1	Node address
Function	1	Function code
Data	n	n is the number of data bytes, it depends on function
CRC	2	Cyclic redundancy check (CRC-16-IBM)

Example of frame in hexadecimal: **01 04 02 FF FF B8 80** (CRC-16-ANSI calculation from 01 to FF gives 80B8, which is transmitted least significant byte first).

1.5.2 Modbus ASCII frame format

Name	Length (bytes)	Description
Start	1	Starts with colon : (ASCII hex value is 3A)
Address	2	Node address in hex
Function	2	Function code in hex
Data	n x 2	n is the number of data bytes, it depends on function
LRC	2	Checksum (Longitudinal redundancy check)
End	2	CR/LF

Address, function, data, and LRC are all capital hexadecimal readable pairs of characters representing 8-bit values (0–255). For example, 122 (7 × 16 + 10) will be represented as 7A.

1.5.3 Modbus TCP frame

Name	Length (bytes)	Description
Transaction ID	2	For synchronization between messages of server and client
Protocol ID	2	0 for Modbus/TCP
Length	2	Number of remaining bytes in this frame
Unit ID	1	Node address
Function	1	Function code
Data	n	n is the number of data bytes, it depends on function

Unit identifier is used with Modbus/TCP devices that are composites of several Modbus devices, e.g. on Modbus/TCP to Modbus RTU gateways. In such case, the unit identifier tells the Slave Address of the device behind the gateway. Natively Modbus/TCP-capable devices usually ignore the Unit Identifier.

1.6 Function Codes

Modbus protocol defines several function codes for accessing Modbus registers. There are four different data blocks defined by Modbus, and the addresses or register numbers in each of those overlap. Therefore, a complete definition of where to find a piece of data requires both the address (or register number) and function code (or register type).

The function codes most commonly recognized by Modbus devices are indicated in the table below. This is only a subset of the codes available - several of the codes have special applications that most often do not apply.

Function Code	Register Type
1	Read Coil
2	Read Discrete Input

3	Read Holding Registers
4	Read Input Registers
5	Write Single Coil
6	Write Single Holding Register
15	Write Multiple Coils
16	Write Multiple Holding Registers

2. Roboteq Implementation

Roboteq's implementation of Modbus doesn't contain all supported functions and modes but contains only subset of it. I this section we are introducing the supported modes and functions implemented in Roboteq's micro controllers.

2.1 Supported Functions

Controllers only supports two functions:

2.1.1 Read Input Registers (0x04)

This function is implemented to read **exactly 4 bytes (2 registers)**. Issuing any messages to read other than 2 registers will return no response.

For examples, to read **VAR1**, you need to read 2 registers from address 0x20C1 so you need to send the following RTU message:

```
01 04 20 C1 00 02 2B F7
```

Name	Description
01	Node address
04	Function code (Read Input Registers)
20 C1	Register address for reading VAR1
00 02	Length of registers to be read (must be 2)
2B F7	Cyclic redundancy check (CRC-16-IBM)

The response for this message will be as following:

```
01 04 04 00 00 12 34 F6 F3
```

Name	Description
01	Node address
04	Function code (Read Input Registers)
04	Total bytes read (always 4 bytes)

00 00 12 34	Value in big Indian notation (MSB first).
F6 F3	Cyclic redundancy check (CRC-16-IBM)

2.1.2 Write Multiple Holding Registers (0x10)

This function is implemented to write **exactly 4 bytes (2 registers)**. Issuing any messages to write other than 2 registers will have no effect.

For examples, to write 0x00001234 to **VAR1**, you need to write 2 registers to address 0x00A1 so you need to send the following RTU message:

```
01 10 00 A1 00 02 04 00 00 12 34 35 6C
```

Name	Description
01	Node address
10	Function code (Write Multiple Holding Registers)
00 A1	Register address for writing VAR1
00 02	Number of registers to write (must be 2)
04	Number of bytes to be written (must be 4)
00 00 12 34	Value to be written in big Indian notation (MSB first)
35 6C	Cyclic redundancy check (CRC-16-IBM)

The response for this message will be as following:

```
01 10 00 A1 00 02 10 2A
```

Name	Description
01	Node address
10	Function code (Write Multiple Holding Registers)
00 A1	Address of written register (VAR1).
00 02	Number of registers written.
10 2A	Cyclic redundancy check (CRC-16-IBM)

2.1.3 Exception responses

Following a request, there are 4 possible outcomes from the slave:

- The request is successfully processed by the slave and a valid response is sent.
- The request is not received by the slave therefore no response is sent.
- The request is received by the slave with a parity, CRC or LRC error (The slave ignores the request and sends no response).

- The request is received without an error, but cannot be processed by the slave for another reason. The slave replies with an exception response.

Here is an example of an exception response:

0A 81 02 B053

0A 81 02 B053

Name	Description
0A	Node address
81	Function code with the highest bit set.
02	The exception code.
B0 53	Cyclic redundancy check (CRC-16-IBM)

The exception codes as explained in the Modbus specification are:

Code	Name	Meaning
0x01	Illegal Function	The function code received in the query is not an allowable action for the slave. This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the slave is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values. If a Poll Program Complete command was issued, this code indicates that no program function preceded it.
0x02	Illegal Data Address	The data address received in the query is not an allowable address for the slave. More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, a request with offset 96 and length 4 would succeed, a request with offset 96 and length 5 will generate exception 02.
0x03	Illegal Data Value	A value contained in the query data field is not an allowable value for the slave. This indicates a fault in the structure of remainder of a complex request, such as that the implied length is incorrect. It specifically does NOT mean that a data item submitted for storage in a register has a value outside the expectation of the application program, since the MODBUS protocol is unaware of the significance of any particular value of any particular register.
0x04	Slave Device Failure	An unrecoverable error occurred while the slave was attempting to perform the requested action.
0x05	Acknowledge	Specialized use in conjunction with programming commands. The slave has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to prevent a

		timeout error from occurring in the master. The master can next issue a Poll Program Complete message to determine if processing is completed.
0x06	Slave Device Busy	Specialized use in conjunction with programming commands. The slave is engaged in processing a long-duration program command. The master should retransmit the message later when the slave is free..
0x07	Negative Acknowledge	The slave cannot perform the program function received in the query. This code is returned for an unsuccessful programming request using function code 13 or 14 decimal. The master should request diagnostic or error information from the slave.
0x08	Memory Parity Error	Specialized use in conjunction with function codes 20 and 21 and reference type 6, to indicate that the extended file area failed to pass a consistency check. The slave attempted to read extended memory or record file, but detected a parity error in memory. The master can retry the request, but service may be required on the slave device.
0x0A	Gateway Path Unavailable	Specialized use in conjunction with gateways, indicates that the gateway was unable to allocate an internal communication path from the input port to the output port for processing the request. Usually means the gateway is misconfigured or overloaded.
0x0B	Gateway Target Device Failed to Respond	Specialized use in conjunction with gateways, indicates that no response was obtained from the target device. Usually means that the device is not present on the network.

2.2 Supported Modes

Controllers are supporting the following modes:

2.2.1 Modbus RTU over TCP (2)

Simply put, this is a Modbus RTU message transmitted with a TCP/IP wrapper and sent over a network instead of serial lines.

For examples, to read **VAR1**, you need to read 2 registers from address 0x20C1 so you need to send the following RTU message:

```
01 04 20 C1 00 02 2B F7
```

Name	Description
01	Node address
04	Function code (Read Input Registers)
20 C1	Register address for reading VAR1
00 02	Length of registers to be read (must be 2)

2B F7	Cyclic redundancy check (CRC-16-IBM)

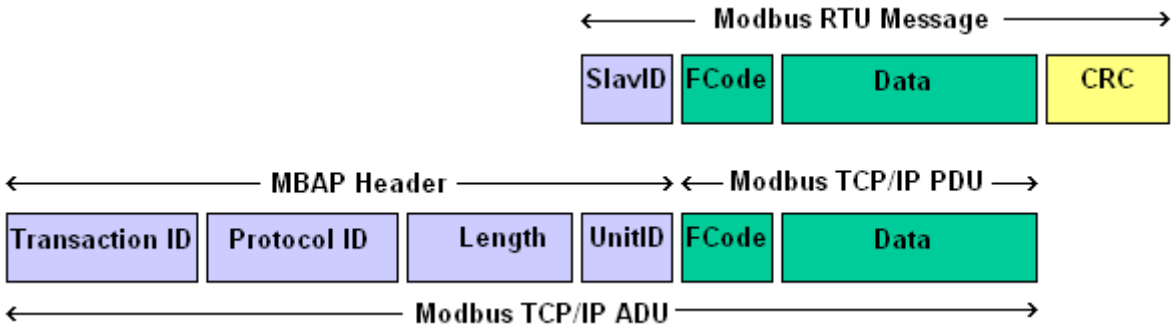
The response for this message will be as following:

```
01 04 04 00 00 12 34 F6 F3
```

Name	Description
01	Node address
04	Function code (Read Input Registers)
04	Total bytes read (always 4 bytes)
00 00 12 34	Value in big Indian notation (MSB first).
F6 F3	Cyclic redundancy check (CRC-16-IBM)

2.2.2 Modbus TCP (1)

Modbus TCP message is the same as RTU over TCP message by removing CRC and adding MBAP header (Modbus Application Header) is being added to the start of the message. Also, node address moved from into MBAP header and named Unit ID.



The MBAP header is consisting from the following:

Name	Description
Transaction ID	2 bytes set by the Client to uniquely identify each request. These bytes are echoed by the Server since its responses may not be received in the same order as the requests.
Protocol Identifier	2 bytes set by the Client, must be 0x0000.
Length	2 bytes identifying the number of bytes in the message to follow.
Unit Identifier	Node address.

For examples, to read **VAR1**, you need to read 2 registers from address 0x20C1 so you need to send the following TCP message:

```
00 03 00 00 00 06 01 04 20 C1 00 02
```

Name	Description
00 03	Transaction ID.
00 00	Protocol Identifier (0x0000 for TCP).
00 06	Number of bytes in the record.
01	Node address
04	Function code (Read Input Registers)
20 C1	Register address for reading VAR1
00 02	Length of registers to be read (must be 2)

The response for this message will be as following:

```
00 03 00 00 00 0D 01 04 04 00 00 12 34
```

Name	Description
00 03	Transaction ID.
00 00	Protocol Identifier (0x0000 for TCP).
00 0D	Number of bytes in the record (13 bytes).
01	Node address
04	Function code (Read Input Registers)
04	Total bytes read (always 4 bytes)
00 00 12 34	Value in big Indian notation (MSB first).

2.2.3 Modbus RS232/RS485 ASCII

Modbus ASCII marks the start of each message with a colon character ":" (hex 3A). The end of each message is terminated with the carriage return and line feed characters (hex 0D and 0A).

In Modbus ASCII, each data byte is split into the two bytes representing the two ASCII characters in the Hexadecimal value.

Modbus ASCII is terminated with an error checking byte called an LRC or Longitudinal Redundancy Check (See appendix B).

For examples, to read **VAR1**, you need to read 2 registers from address 0x20C1 so you need to send the following ASCII message:

```
:0104 20C1 0002 18<CRLF>
```

Name	Description
':'	Start of message - 0x3A

'0' '1'	Node address – 0x01
'0' '4'	Function code (Read Input Registers) – 0x04
'2' '0' 'C' '1'	Register address for reading VAR1 – 0x20C1
'0' '0' '0' '2'	Length of registers to be read (must be 2) – 0x0002
'1' '8'	LRC
<CRLF>	End of message, carriage return and line feed – 0x0D0A

The response for this message will be as following:

:01040400001234DE<CRLF>

Name	Description
':'	Start of message - 0x3A
'0' '1'	Node address – 0x01
'0' '4'	Function code (Read Input Registers) – 0x04
'0' '4'	Read data length (4 bytes) – 0x04
'0' '0' '0' '0' '1' '2' '3' '4'	Value read from VAR1 – 0x00001234
'D' 'E'	LRC
<CRLF>	End of message, carriage return and line feed – 0x0D0A

2.3 Register Address Calculation

Register address is calculated based on CANOpen ID of the command/query. You can use the following steps to get the register address.

- From Roboteq Manual find your command/query.
- Get the CANOpenID value.
- Left shift the CANOpenID with 5 bits.
- OR the result with the command/query index.
- AND the result with 0xFFFF.
- The resulting value will be the register address.

For example, the read user integer variable CANOpen ID is 0x2106 and suppose we are required to read the first variable:

- CANOpenID = 0x2106
- Left shift with 5 → 0x2106 << 5 = 0x420C0
- OR with index → 0x420C0 | 0x01 = 0x420C1
- AND the result with 0xFFFF → 0x420C1 & 0xFFFF = 0x20C1
- Use 0x20C1 as the address.

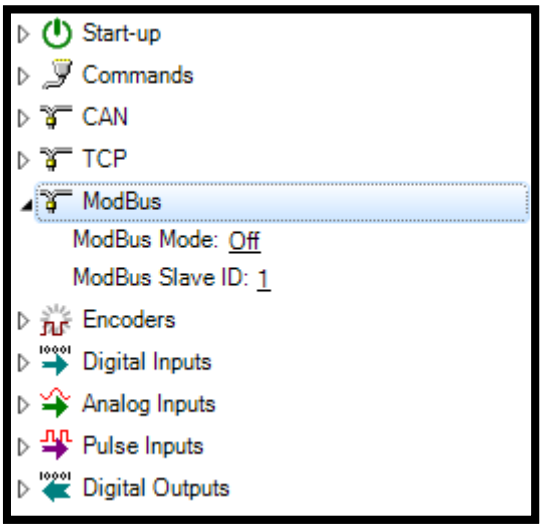
You can also calculate the register address from the tables in the appendix A, you need to get the Modbus ID value from the table then add to it the desired command/query index.

For example, the read user integer variable Modbus ID is 0x20C0, to get the first variable:

- ModbusID = 0x20C0.
- Add the index → 0x20C0 + 0x01 = 0x20C1
- Use 0x20C1 as the address.

2.4 Controller Configuration

To configure controller to use Modbus, you will find new configuration section called Modbus under control board:



From the configuration you could set Modbus mode to (Off, TCP, RTU over TCP, RS232 ASCII, or RS485 ASCII). You will be also able to set the Modbus Save ID.

You can use terminal to issue commands for changing mode and slave ID:

Command	Description
^DMOD 0	Set mode Off.
^DMOD 1	Set mode TCP.
^DMOD 2	Set mode RTU over TCP.
^DMOD 3	Set mode RS232 ASCII.
^DMOD 4	Set mode RS485 ASCII.
^MNOD 5	Set slave ID to 5.
~DMOD	Query mode.
~MNOD	Query slave ID.

3. Appendix A: Commands Mapping

3.1 Command Mapping

Command	CANOpen ID	Modbus ID (Hex)	Modbus ID (Dec)
P	0x2001	0x0020	32
S	0x2002	0x0040	64
C	0x2003	0x0060	96
CB	0x2004	0x0080	128
VAR	0x2005	0x00A0	160
AC	0x2006	0x00C0	192
DC	0x2007	0x00E0	224
DS	0x2008	0x0100	256
D1	0x2009	0x0120	288
D0	0x200A	0x0140	320
R	0x2018	0x0300	768
H	0x200B	0x0160	352
EX	0x200C	0x0180	384
MG	0x200D	0x01A0	416
MS	0x200E	0x01C0	448
PR	0x200F	0x01E0	480
PX	0x2010	0x0200	512
PRX	0x2011	0x0220	544
AX	0x2012	0x0240	576
DX	0x2013	0x0260	608
B	0x2015	0x02A0	672
SX	0x2014	0x0280	640
CG	0x2000	0x0000	0
RC	0x2016	0x02C0	704
EES	0x2017	0x02E0	736
AO	0x2019	0x0320	800
TX	0x201A	0x0340	832
TV	0x201B	0x0360	864
CSW	0x201C	0x0380	896
PSW	0x201D	0x03A0	928
ASW	0x201E	0x03C0	960

3.2 Query Mapping

Query	CANOpen ID	Modbus ID (Hex)	Modbus ID (Dec)
A	0x2100	0x2000	8192
M	0x2101	0x2020	8224
P	0x2102	0x2040	8256
S	0x2103	0x2060	8288
C	0x2104	0x2080	8320

CB	0x2105	0x20A0	8352
VAR	0x2106	0x20C0	8384
SR	0x2107	0x20E0	8416
CR	0x2108	0x2100	8448
BCR	0x2109	0x2120	8480
BS	0x210A	0x2140	8512
BSR	0x210B	0x2160	8544
BA	0x210C	0x2180	8576
V	0x210D	0x21A0	8608
D	0x210E	0x21C0	8640
DI	0x6400	0x8000	32768
AI	0x6401	0x8020	32800
PI	0x6403	0x8060	32864
T	0x210F	0x21E0	8672
F	0x2110	0x2200	8704
FS	0x2111	0x2220	8736
FF	0x2112	0x2240	8768
B	0x2115	0x22A0	8864
DO	0x2113	0x2260	8800
E	0x2114	0x2280	8832
CIS	0x2116	0x22C0	8896
CIA	0x2117	0x22E0	8928
CIP	0x2118	0x2300	8960
TM	0x2119	0x2320	8992
LK	0x2124	0x2480	9344
TR	0x2125	0x24A0	9376
K	0x211A	0x2340	9024
DR	0x211B	0x2360	9056
AIC	0x6402	0x8040	32832
PIC	0x6404	0x8080	32896
MA	0x211C	0x2380	9088
MGD	0x211D	0x23A0	9120
MGT	0x211E	0x23C0	9152
MGM	0x211F	0x23E0	9184
MGS	0x2120	0x2400	9216
MGY	0x2121	0x2420	9248
FM	0x2122	0x2440	9280
HS	0x2123	0x2460	9312
QO	0x2126	0x24C0	9408
EO	0x2127	0x24E0	9440
RMA	0x2128	0x2500	9472
RMG	0x2129	0x2520	9504
RMM	0x212A	0x2540	9536

ML	0x212B	0x2560	9568
TS	0x212C	0x2580	9600
MRS	0x212D	0x25A0	9632
MZ	0x212E	0x25C0	9664
PK	0x212F	0x25E0	9696
RF	0x2130	0x2600	9728
FIN	0x2137	0x26E0	9952
GY	0x2131	0x2620	9760
ANG	0x2132	0x2640	9792
SCC	0x2133	0x2660	9824
ICL	0x2134	0x2680	9856
FC	0x2135	0x26A0	9888
SL	0x2136	0x26C0	9920

4. Appendix B: CRC Calculation

```
const u8 modbus_crc_hi[] = {
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,

       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

       0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
       0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
```

```
        0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

        0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,

        0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
        0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
};

const u8 modbus_crc_lo[] = {
        0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2,
        0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04,

        0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E,
        0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09, 0x08, 0xC8,

        0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A,
        0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC,

        0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6,
        0xD2, 0x12, 0x13, 0xD3, 0x11, 0xD1, 0xD0, 0x10,

        0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32,
        0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,

        0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE,
        0xFA, 0x3A, 0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38,

        0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA,
        0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C,

        0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
        0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0,

        0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62,
        0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4,

        0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F, 0x6E, 0xAE,
        0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,

        0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA,
        0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C,

        0xB4, 0x74, 0x75, 0xB5, 0x77, 0xB7, 0xB6, 0x76,
        0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0,

        0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92,
        0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54,

        0x9C, 0x5C, 0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E,
        0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98,

        0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A,
        0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,

        0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86,
        0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80, 0x40
};
```

```
u16 modbus_calcCRC(const u8 *p, int length) {
    u8 hi = 0xFF;
    u8 lo = 0xFF;

    while (length--) {
        u8 i = lo ^ *p++;
        lo = hi ^ modbus_crc_hi[i];
        hi = modbus_crc_lo[i];
    }
    return (u16) (hi << 8) | lo;
}


u8 modbus_calcLRC(u8* inputBuffer, int length)
{
     u8 sum = 0;
     for(int i = 0; i < length; i++)
     {
         sum += inputBuffer[i];
     }

     return ~sum + 1;
}
```