Modbus Compact

Joachim Bürmann, *IFTOOLS GmbH*

May 24, 2023

The Modbus is a multi-drop network based communication protocol for master/client architecture. Originally published by Modicon (now Schneider electric) in 1979 to communicate with PLCs (programmable logic controller) it has become a de facto standard for its simplicity and robustness. Serial Modbus connections provide two basic transmission modes: ASCII and RTU. The Modbus/TCP Security protocol (based on TLS) is not discussed in this article.

Overview

Modbus is a Master/Slave protocol. Only one master is connected to the bus at the same time. A conversation always starts with a request from the master, (a slave never starts a transmission on its own behalf). The master sends a message and depending on the message content, a slave responses to it. Each Modbus message has the same structure and is composed of the same four field. The sequence of the fields is always the same, thus makes it easy to parse and evaluate the messages by the bus participants.

Device address	Address of the message receiver
Function code	Code defining the message type
Data	Data block with additional information
Checksum	Checksum of the message

The address field in the message is used to specify the device which should response to the request. All other devices with a different address ignore the message.
The requested device responses with the same address (not the master address) which we will see later may cause some difficulties when determining the message sender by an outside observer.

The function code (or number) specifies the meaning of the following data. The last field in a Modbus message or telegram is a checksum to validate the correct transmission of the message content.

Modbus transmission modes

Modbus over serial line provides two basic transmission modes, ASCII or RTU. In ASCII messages consists of readable ASCII characters whereas RTU transmits the data in raw binary format, makes the messages unreadable when simply monitoring it, but reduces the message size to a minimum.
To give the bus participants an easy way to detect the start and end of a telegram transmission, Modbus messages are framed either by a start/end sequence (ASCII) or a defined idle/pause time (RTU).
In Modbus ASCII every message starts with a colon ':' and ends with a carriage return linefeed (CRLF). In Modbus RTU telegrams are separated by a transmission pause (idle time) of at least 3.5 characters (the time it needs to send 3.5 bytes). The two modes in detail:

Modbus ASCII

In a Modbus ASCII communication each 8-bit byte in a telegram is represented by two ASCII characters in the range of 0-9 and A-F (hexadecimal digits). For instance the byte value hex 5B is encoded as the two characters 0x35 ("5") and 0x42 ("B").
To separate the telegrams every telegram starts with a colon and ends with a carriage return line feed. Since these characters are not part of the hexadecimal character range, they cannot mixed up with the actual data and are easily detectable as a telegram start or end. In

contrary to the RTU mode Modbus ASCII makes no special timing requirements. This becomes especially important when you transmit data over a medium with very low time allowance like a modem connection.
The downside is, that all data bytes must be sent as pairs of hexadecimal characters encoded in ASCII. This makes the protocol more human readable but means the double size of data which must be transmitted over the line. And the sending and receiving applications must convert the raw data to and from ASCII.

Telegram frame

A Modbus ASCII frame looks like:

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0...2*252 char(s)	2 chars	2 chars CR,LF

Timing

Modbus ASCII is not very affected by timings. Unless the user has configured a longer timeout, intervals of up to one second between every sent character are allowed. But even timeouts of several seconds are not uncommon in wide area networks.

LRC Checksum

Each telegram in ASCII mode includes a checksum field. The checksum itself based on a Longitudinal Redundancy Checking (LRC) calculation which is performed on the telegram body (excluding the starting colon, the ending CRLF and - of course - the checksum field).
An according checksum algorithm in Lua is:

```lua
function LRC( data )
  local sum = 0
  for i=1,#data-1 do
    sum = sum + data:byte(i)
  end
  return bit.band( 0-sum, 0xFF );
end
```

Modbus RTU

In the RTU mode the data is transmitted in 8-bit values (raw binary sequences). There is also no special byte value indicating the beginning or ending of a telegram sequence. This makes Modbus RTU telegrams very compact in size. But you need another mechanism to detect the start of every telegram in the data stream, see section Modbus RTU Timing.

Telegram frame

A Modbus RTU frame looks like:

Address	Function	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes

The maximum size of a Modbus RTU frame therefore is 256 bytes.

Timing

Modbus RTU Timing Modbus RTU does not define any start and/or end sequence. In Modbus RTU telegram bytes must be sent consecutively with a maximum delay of 1.5 characters in between them and with a specified minimum 3.5 character space between the telegrams as a delimiter. A character here is defined as the time it needs to send a character or byte with the given baud rate. For example: A transmission format of 11 bits (1 start bit, 8 data bits, 1 parity bit and 1 stop bit) and a transmission rate of 19200 baud gives you:

$$1.5 * \frac{11}{19200} \approx 0.00086s \quad \text{and} \quad 3.5 * \frac{11}{19200} \approx 0.002s$$

It is worth mentioning that a lot of Modbus RTU problems are caused by violation of these timing specifications. And: The exact values for the telegram delimiter may differ in certain applications.

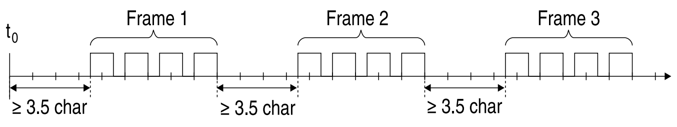

Figure 1: *Modbus RTU frame timing*

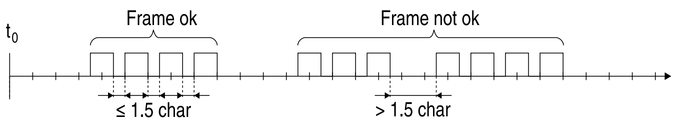

Figure 2: *Modbus RTU byte timing*

Remark

In [1] it is recommended to use a fixed timeout for both timings $t_{1.5}$ and $t_{3.5}$ when using baud rates above 19200. This especially to avoid heavy CPU loads in the bus participants. The recommendations are:
$t_{1.5} = 750\mu s$ for the inter-character timeout and
$t_{3.5} = 1.750ms$ for the inter-frame delay.

CRC Checksum

Telegrams in RTU mode contain an error checking field based on a Cyclical Redundancy Checking (CRC) method. The CRC checksum is a 16 bit value calculated from the entire message (except for the CRC field itself). This means: Address, function and data.
The 16 bit checksum is added to the message as the last two bytes whereas the low byte is added first.
The following code snippet shows a CRC16 calculation for Modbus (polynomial 0xA001) implemented in Lua.

```lua
function crc16( data )
    local crcFull=0xFFFF
    local poly=0xA001
    for i=1,#data do
      crcFull=bit.bxor(crcFull,data:byte(i))
      for j=1,8 do
         crclsb=bit.band(crcFull,0x0001)
         crcFull=bit.rshift(crcFull,1)
         crcFull=bit.band(crcFull,0x7FFF)
         if crclsb == 1 then
           crcFull=bit.bxor(crcFull,poly)
         end
      end
    end
    local msb=bit.rshift(crcFull,8)
    local lsb=bit.band(crcFull,0xFF)
    return msb*256+lsb
end
```

Modbus addressing rules

Modbus uses a 8-bit value to address the recipient of a message. In Modbus ASCII it is coded as two hexadecimal characters, in Modbus RTU as a single byte. The available address range is divided as:

Address	Description
0	Broadcast address, accepted by all devices
1...247	Individual Modbus devices
248...255	Reserved

Since a Modbus message always only contains the address of the requested device (even in the device response), there is no special need for a master address.

The Modbus data model

Modbus was originally designed as a protocol to access and control several PLCs[1] over a network. PLCs replaced the hard-wired automation control of those days and came with a far more flexible and in particular programmable input and output (IO) conception.

Typical PLCs range from small modules with a few IOs to complex modular devices with thousands IOs, register memory and sometimes with an integral processor. PLCs provide access to the several IOs and control register via a memory based model, whereas IOs can be divided roughly in the four topics: analog inputs, digital inputs, analog outputs and digital outputs. Additional further memory registers are used for controlling or holding parameters (e.g. like full scales).
Modbus adopt this concept in its own design. In Modbus every accessible data is organized too in four general data banks or address segments. These are in Modbus terminology: coils, discrete inputs, holding registers and input registers. The names may vary in applications. Holding registers are sometimes called output register, coils referred as digital or discrete (bitwise) outputs[2].
The following table shows the relationship between the several IO types and according Modbus functions.

Type & Address	**Access**	**Modbus Function Name**
Discrete Output Data type: Single bit Address: 1-9999	Read	Read Coils (01)
	Write	Write Single Coil (05)
Holding Register Data type: 16-bit word Address: 40001-49999	Read	Read Holding Register (03)
	Write	Write Single Register (06)
Discrete Input Data type: Single bit Address: 10001-19999	Read	Read Discrete Input (02)
Input Register Data type: 16-bit word Address: 30001-39999	Read	Read Input Registers (04)

Table 1: *IO Function Overview*

Discrete inputs and outputs are mostly digital IOs, input registers may be referred as analog inputs.
Holding registers are the most universal 16-bit register in Modbus. They can be read and written and therefore be used for general usage like inputs, outputs, configuration data and more. As their name indicated, they just can 'holding' data.

The address ranges in the left column are a relic of early Modbus days and often referred to as the old Modicon convention. You neither will see nor use these address offsets in a Modbus sequence. But they may appear in older device specifications and serves there to distinguish different Modbus IO types or banks.
For example: If a register documentation tells you the register number is 40001 it means the holding (analog

[1] programmable logic controller

[2] Coils are named after the coils in relays

output) register 1. If a register is referred as 30002, it is (analog) input register number 2. The address therefore specifies not only the location but also how to access the register without naming the type.

Modbus Function Codes

Modbus functions are specified by the function number or code in the telegram. Function codes are like commands to access (amongst others) one of the four general IO types (banks or address ranges) described above.
Beside these basic functions for data access exist further function codes for diagnostic, file record access and to transport other protocol sequence in a Modbus telegram (encapsulated interface transport).
The functions supported by a Modbus device are vendor independent and may only provide a small set of the following list.

Function	Description
01	Read Coils
02	Read Discrete Inputs
03	Read Holding Register
04	Read Input Registers
05	Write Single Coil
06	Write Single Register
07	Read Exception Status (Serial Line only)
08	Diagnostics (Serial Line only)
11	Get Comm Event Counter (Serial Line only)
12	Get Comm Event Log (Serial Line only)
15	Write Multiple Coils
16	Write Multiple Registers
17	Report Slave ID
20	Read File Record
21	Write File Record
22	Mask Write Register
23	Read/Write Multiple Registers
24	Read FIFO Queue
43	Encapsulated Interface Transport

Table 2: *Common Modbus function codes*

01 (0x01) Read Coils

Function to read from 1 up to 2000 continuous states of digital outputs (coils). The request specifies the start address of the first coil and the number of coils to be read.
The output states are packed as one coil per bit in the response data field from low (LSB) to high order (MSB). The coil state is indicated as 1=ON, 0=OFF. If the requested number of coils is not a multiple of eight, the last byte in the response is padded with zeros.
The byte count field in the response specifies the number of bytes needed to hold the data.

Request

Function code	1 Byte	**0x01**
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of coils	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x01**
Byte count	1 Byte	N
Coil status	n Bytes	`n = N or N + 1`

Error

Function code	1 Byte	**0x81**
Exception code	1 Byte	1, 2, 3, 4

Example

Read of the digital output states 2 to 13 from device 7. All together 12 states which are returned in two bytes (with 4 padding zeros).
Please note! The address of the output states counts from 0, coil number 2 is therefore indexed with 1.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**07**	Address	**07**
Function	**01**	Function	**01**
Address High Byte	**00**	Byte Count	**2**
Address Low Byte	**01**	Output 2-9	**FF**
Quantity High Byte	**00**	Output 10-13	**0F**
Quantity Low Byte	**0C**	CRC16 low	**30**
CRC16 low	**6D**	CRC16 high	**08**
CRC16 high	**A9**		

In the example the responded state of all 12 coils is ON, which is: 11111111 00001111 or FF_h $0F_h$. The remaining bits 13-16 are padded with zero.

02 (0x02) Read Discrete Inputs

This function is used to read the state of digital inputs. It is similar to the Read Coils function but addresses digital inputs instead.
The function reads from 1 to 2000 continuous states of digital inputs. The request specifies the start address (count from zero) of the first input and the number of inputs to be read.
The results are packed as one digital input per bit in the response data field from low (LSB) to high order

(MSB). The input state is indicated as 1=ON, 0=OFF. The byte count field in the response specifies the number of bytes needed to hold the data. If the requested quantity of inputs is not a multiple of eight, the last byte in the response is padded with zeros.

Request

Function code	1 Byte	**0x02**
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of inputs	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x02**
Byte count	1 Byte	N
Input status	n Bytes	`n = N or N + 1`

Error

Function code	1 Byte	**0x82**
Exception code	1 Byte	1, 2, 3, 4

Example

Read the digital input states 4 to 23 from device 17. The result is covered in 3 bytes (20 inputs).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**11**	Address	**11**
Function	**02**	Function	**02**
Address High Byte	**00**	Byte Count	**03**
Address Low Byte	**03**	Output 4-11	**FF**
Quantity High Byte	**00**	Output 12-19	**FF**
Quantity Low Byte	**14**	Output 20-23	**0F**
CRC16 low	**88**	CRC16 low	**4B**
CRC16 high	**05**	CRC16 high	**1A**

The 20 inputs are transmitted as 3 bytes, here all inputs are ON. The first two bytes are 11111111 or FF_h, the third byte contains the last 4 bits and a zero padding for the remaining four bits 00001111 or $0F_h$.

03 (0x03) Read Holding Register

In Modbus holding registers are the most universal register type. You can think of them as a general 16-bit memory, used as analog output, input or configuration register holding a certain value.
Like in the former Read Coils and Read Discrete Inputs the function specifies a start address and a quantity to read a continuous block of holding registers. The address counts from zero. A register itself is a 16-bit value. Modbus itself does not specify the kind of data 'behind' a register. Originally the values were limited to 16-bit words, but newer devices may also provide 32-bit floating-point values represented by two successive 16-bit registers. But this is application specific and not part of this article.

Request

Function code	1 Byte	**0x03**
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x N
Register value	N x 2 Bytes	

Error

Function code	1 Byte	**0x83**
Exception code	1 Byte	1, 2, 3, 4

Example

The following example shows a request to read the registers 200 - 201 from device 240 (F0$_h$).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**F0**	Address	**F0**
Function	**03**	Function	**03**
Address High Byte	**00**	Byte Count	**04**
Address Low Byte	**C7**	Register (200) High Byte	**43**
Quantity High Byte	**00**	Register (200) Low Byte	**67**
Quantity Low Byte	**02**	Register (201) High Byte	**B3**
CRC16 low	**A1**	Register (201) Low Byte	**33**
CRC16 high	**17**	CRC16 low	**8B**
		CRC16 high	**82**

The slave returns as results for register 200 4367$_h$ or 17255, for register 201 B333$_h$ or 45875. The latter may be a two-complementary value and therefore negative. Both register may also form a 32-bit floating-point value 4367B333$_h$ which is the binary representation of 231.7. As mentioned before: This interpretation of the registers depends on the device and application and is not further considered by this article.

04 (0x04) Read Input Registers

Input registers are equal to analog inputs and specified as 16-bit words. Again: The meaning of a single

or successive register values is application dependent. With the IFTOOLS analyzer software you can format the output depending on a certain device (address) and/or register. But from the Modbus protocol point of view they are 16-bit numbers.
The address counts from zero, the register values are in the big endian order, means: high byte followed by low byte.

Request

Function code	1 Byte	**0x04**
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x04**
Byte count	1 Byte	2 x N
Register value	N x 2 Bytes	

Error

Function code	1 Byte	**0x84**
Exception code	1 Byte	1, 2, 3, 4

Example

The Read Input Registers function follows the same rules as described in Read Holding Register. Here a small example which reads two holding registers (analog inputs) at address 0 from device 1.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**01**	Address	**01**
Function	**04**	Function	**04**
Address High Byte	**00**	Byte Count	**04**
Address Low Byte	**00**	Register (0) High Byte	**00**
Quantity High Byte	**00**	Register (0) Low Byte	**00**
Quantity Low Byte	**02**	Register (1) High Byte	**00**
CRC16 low	**71**	Register (1) Low Byte	**00**
CRC16 high	**CB**	CRC16 low	**84**
		CRC16 high	**FB**

The result of both registers is 0000_h.

05 (0x05) Write Single Coil

A single coil means a single digital output (coils comes from the coils used in older relays). This function is intended to set a single digital output in a remote device to ON or OFF. A value of $FF00_h$ means ON, a value of 0000_h means OFF. Without an error the response is the echo of the request.

Coil addresses are counted from zero. So Coil number 1 is addressed as 0.

Request

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Response

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Error

Function code	1 Byte	**0x85**
Exception code	1 Byte	1, 2, 3, 4

Example

The example switches the digital output at output address 7 to ON in the remote device 1. Remember: Coil addresses starting at zero. Address 6 in the example therefore means Coil (or digital output) 7.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**01**	Address	**01**
Function	**05**	Function	**05**
Output Address High	**00**	Output Address High	**00**
Output Address Low	**06**	Output Address Low	**06**
Output Value High	**FF**	Output Value High	**FF**
Output Value Low	**00**	Output Value Low	**00**
CRC16 low	**6C**	CRC16 low	**6C**
CRC16 high	**3B**	CRC16 high	**3B**

06 (0x06) Write Single Register

This function is used to write a value into a single holding register of a remote device. The request specifies the register address counting from zero and a 16-bit word as the output value.
Writing data in successive registers, for instance a 32-bit floating-pint number or other values bigger as 16-bit, is provided by the Modbus function Write Multiple Registers.

Request

Function code	1 Byte	**0x06**
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Response

Here again the normal response is just the echo of the request.

Function code	1 Byte	**0x06**
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 or 0xFFFF

Error

Function code	1 Byte	**0x86**
Exception code	1 Byte	1, 2, 3, 4

Example

Here is an example of a request to write holding register 6 to 0019$_h$ or 25 decimal in the remote device 128 (address 80$_h$). In case of no error the slave echoes the request back as response.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**80**	Address	**80**
Function	**06**	Function	**06**
Register Address High	**00**	Register Address High	**00**
Register Address Low	**05**	Register Address Low	**05**
Register Value High	**00**	Register Value High	**00**
Register Value Low	**19**	Register Value Low	**19**
CRC16 low	**46**	CRC16 low	**46**
CRC16 high	**10**	CRC16 high	**10**

07 (0x07) Read Exception Status (Serial line only)

Reads the contents of eight exception status coils within a slave. The meaning of the eight coils is user-defined, their addresses controller dependent.
Typical applications are to hold information about the controller's status. For example: Machine on/off, battery state, error conditions or other user defined flags.
Broadcast is not supported!
The function provides a simple and by its short message length a fast method for accessing this information, because the Exception Coil references are known by the device (no coil reference is needed in the function).
Please note! The Modbus specification defines this as a serial only function. Therefore it should only be used if the addressed slave is a serial device.

Request

Function code	1 Byte	**0x07**

Response

Function code	1 Byte	**0x07**
Output Data	1 Byte	0x00 to 0xFF

Error

Function code	1 Byte	**0x87**
Exception code	1 Byte	1 or 4

Example

Here is an example of a request to read the exception status in slave device 19.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**13**	Address	**13**
Function	**07**	Function	**07**
CRC16 low	**4D**	Output Data	**6D**
CRC16 high	**42**	CRC16 low	**43**
		CRC16 high	**D8**

The returned exception status (Output Data) is 6D$_h$ or 01101101. Left to the right (highest bit to lowest bit) the output is: OFF-ON-ON-OFF-ON-ON-OFF-ON. The status itself is shown from the (device internal) highest to the lowest addressed coil.

08 (0x08) Diagnostics (Serial line only)

Modbus provides this special function to perform some communication tests between the master and a slave or to check internal error conditions in a slave device. The function is limited to serial devices.
The Diagnostics function uses a two-byte sub-function code listed below in the request to specify the kind of test. Some tests also need further data to be send to the test device which is transmitted in the data following the sub-function field.
Normally using the diagnostics function should not effect the remote device (internal state and registers). Nevertheless certain tests can optionally reset error counters in the addressed device.
And: A slave can be forced into a 'Listen Only Mode' in which it only monitors the transmitted messages but does not responses anymore. This clearly has an effect to the application program. Especially if it depends on the responses of the device. However it sometimes becomes useful to remove a malfunctioning device from the bus by putting it in the 'Listen Only Mode'.

Request

Function code	1 Byte	**0x08**
Sub-Function	2 Bytes	
Data	N x 2 Bytes	

Response

Function code	1 Byte	**0x08**
Sub-Function	2 Bytes	
Data	N x 2 Bytes	

Error

Function code	1 Byte	**0x87**
Exception code	1 Byte	1 or 3 or 4

Supported Sub-function Codes

The naming of Modbus devices differs depending on the documentation. Sometimes the master is also called the client, and the slaves named as servers. In this document we prefer the master/slave convention. The following sub-function names therefore speaks of slaves instead of clients.

Sub-Function	Name
0x0000	Return Query Data
0x0001	Restart Communications Option
0x0002	Return Diagnostic Register
0x0003	Change ASCII Input Delimiter
0x0004	Force Listen Only Mode
0x0005...0x0009	RESERVED
0x000A	Clear Counters and Diagnostic Register
0x000B	Return Bus Message Count
0x000C	Return Bus Communication Error Count
0x000D	Return Bus Exception Error Count
0x000E	Return Slave Message Count
0x000F	Return Slave No Response Count
0x0010	Return Slave NAK Count
0x0011	Return Slave Busy Count
0x0012	Return Bus Character Overrun Count
0x0013	RESERVED
0x0014	Clear Overrun Counter and Flag
0x0015...0xFFFF	RESERVED

Table 3: *Modbus Error Codes*

00 Return Query Data

The tested device is forced to return (loop back) the data received in the data field. The response must be identical to the request.

Sub-Function	Data Request	Data Response
00 00	Any Data	Echo Request Data

01 Restart Communications Option

The remote device must re-initialize and restart its serial port and also must clear all its communication event counters. In 'listen Only Mode' no response is expected.
Please note! This is the only method to bring a device back from the 'Listen Only Mode'.
If the device is not in 'Listen Only Mode', it returns a normal response before performing the restart.
Depending on the data field, the device also clears its communication events log (FF 00) or leave it unchanged (00 00).

Sub-Function	Data Request	Data Response
00 01	00 00	Echo Request Data
00 01	FF 00	Echo Request Data

02 Return Diagnostic Register

The Modbus specification supports a 16-bit wide diagnostic register in very slave device. The meaning of the 16 bit values are vendor dependent. Bit 15 is the highest bit. The master can query the content of this register with sub-function 2.

Sub-Function	Data Request	Data Response
00 02	00 00	Diagnostic Register Contents

03 Change ASCII Input Delimiter

In Modbus ASCII Mode, the last character of a message is the linefeed. But sometimes it is not wanted to have the linefeed as message delimiter. With sub-function 02 the linefeed character can be replaced by another one.
Please note! The new character should not occur in other parts of a ASCII message. Thus ':', carriage return and the characters '0'...'9' as also 'A'...'F' are excluded.

Sub-Function	Data Request	Data Response
00 03	CHAR 00	Echo Request Data

04 Force Listen Only Mode

Tells the addressed device to switch into 'Listen Only Mode'. In this mode the device not longer communicates active with the bus but only listen (e.g. to detect a command to end this mode). This is an appropriate method to mute the device, allowing the rest of the bus participants to communicate without interruptions

of this device. The device immediately becomes mute, no response is returned.
The only command accepted by the device, when in 'Listen Only Mode' is the Restart Communications Option (function 8, sub-function 01).

Sub-Function	Data Request	Data Response
00 04	00 00	No response returned

10 Clear Counters and Diagnostic Register

Modbus devices may provide internal counter and diagnostic register. With this function the master can force the addressed device to clear them.

Sub-Function	Data Request	Data Response
00 0A	00 00	Echo request data

11 Return Bus Message Count

Returns the number of all messages the addressed device received after last restart, counter clearing or power-up. Note! This includes not only the messages for the addressed slave but all transmitted messages received on the bus.

Sub-Function	Data Request	Data Response
00 0B	00 00	Total message count

12 Return Bus Communication Error Count

This diagnostics sub-function returns the number of CRC errors the addressed device encountered since its last restart, clear counter instruction or power-up.

Sub-Function	Data Request	Data Response
00 0C	00 00	CRC error count

13 Return Bus Exception Error Count

The function queries the number of Modbus exceptions returned by the device since its last restart, clear counter instruction or power-up. The several exception types are listed in section Modbus Exception Codes.

Sub-Function	Data Request	Data Response
00 0D	00 00	Exception error count

14 Return Slave Message Count

In contrary to sub-function 11 this function only returns the quantity of messages addresses to a given device since its last restart, clear counter instruction or power-up. The number covers also the received broadcast messages.

Sub-Function	Data Request	Data Response
00 0E	00 00	Slave message count

15 Return Slave No Response Count

Queries the quantity of requests the addressed device did not answered since its last restart, clear counter instruction or power-up (neither a normal response nor an exception response).

Sub-Function	Data Request	Data Response
00 0F	00 00	Slave not response count

16 Return Slave NAK Count

The function queries the number of requests addressed to the slave for which it returns a NAK (negative Acknowledge) exception response, since its last restart, clear counters operation, or power–up. The exception responses are listed in section Modbus Exception Codes.

Sub-Function	Data Request	Data Response
00 10	00 00	Slave NAK count

17 Return Slave Busy Count

The function queries the number of requests addressed to the slave for which it returns a Slave Device Busy Exception response, since its last restart, clear counters operation, or power–up. The exception responses are listed in section Modbus Exception Codes.

Sub-Function	Data Request	Data Response
00 11	00 00	Slave Busy count

18 Return Bus Character Overrun Count

The function queries the number of requests addressed to the slave which it could not handled caused by a character overrun condition, since its last restart, clear counters operation, or power–up. A character overrun occurs when the slave device receives characters faster than it can handle them or by a hardware malfunction.

Sub-Function	Data Request	Data Response
00 12	00 00	Slave character overrun count

20 Clear Overrun Counter and Flag

Clears the overrun error counter in the addressed device and reset its error flag.

Sub-Function	Data Request	Data Response
00 20	00 00	Echo request data

Example

Here is an example of a diagnostic test for device 23. The device is asked to echo the data in the request (sub-function 00).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**17**	Address	**17**
Function	**08**	Function	**08**
Sub-Function high	**00**	Sub-function high	**00**
Sub-Function low	**00**	Sub-Function low	**00**
Sub-Function high	**AA**	Sub-function high	**AA**
Sub-Function low	**55**	Sub-Function low	**55**
CRC16 high	**5C**	CRC16 high	**5C**
CRC16 low	**62**	CRC16 low	**62**

11 (0x0B) Get Comm Event Counter (Serial line only)

The event counter in a device is incremented every time a request was successfully handled (responded) by the device. The counter will not increased for exception responses, poll or fetch event counter commands.
This function becomes handy if the master (or application) want to know whether the messages to given slave are all handled correctly. Normally this is done by checking the event counter before and after a series of messages.
The event counter can be reset via the Diagnostic function (0x08) using sub-function 'Restart Communication Option' (00 01).
The function returns beside the event count a status word indicating if the device is busy (already processes a former command, status FFFF_h) or not (status 0000_h).

Request

Function code	1 Byte	**0x0B**

Response

Function code	1 Byte	**0x0B**
Status	2 Bytes	0x0000 or 0xFFFF
Event Count	2 Bytes	0x0000 to 0xFFFF

Error

Function code	1 Byte	**0x8B**
Exception code	1 Byte	1 or 4

Example

Here is an example of a request to read the event count of remote device 35. The device responses with an event count of 264 (0108_h) and a not busy status.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**23**	Address	**23**
Function	**0B**	Function	**0B**
CRC16 low	**59**	Status high	**00**
CRC16 high	**47**	Status low	**00**
		Event Count high	**01**
		Event Count low	**08**
		CRC16 low	**A2**
		CRC16 high	**DF**

12 (0x0C) Get Comm Event Log (Serial line only)

Function 12 is an effective way to gain more statistical information about the communication status of a remote device. This command gives you the busy status, the event count, the message count and a series of bytes containing the status of the last up to 64 send or receive operations handled by the device.
The meaning of the status and event count is already described in Get Comm Event Counter. The message counter contains the number of messages the remote device has processed since its last restart, clear counters operation or power-up. You can query the message counter also with Diagnostics function, sub-function 11 (Return Bus Message Count).

Request

Function code	1 Byte	**0x0C**

Response

Function code	1 Byte	**0x0C**
Byte Count	1 Byte	N*
Status	2 Bytes	0x0000 or 0xFFFF
Event Count	2 Bytes	0x0000 to 0xFFFF
Message Count	2 Bytes	0x0000 to 0xFFFF
Event Bytes	N-6 * Bytes	

Error

Function code	1 Byte	**0x8C**
Exception code	1 Byte	1 or 4

Coding of the event bytes

There are two kind of event bytes distinguished by the highest bit 7. Two further events are coded with bit7 and bit 6 always set to zero.The bits marked as – are not used.

Bits 7 6 5 4 3 2 1 0	**Description**
1 x x x - - x -	Remote device MODBUS Receive Event
1 x x x - - 1 -	Communication error
1 x x 1 - - x -	Character overrun
1 x 1 x - - x -	Currently in Listen Only Mode
1 1 x x - - x -	Broadcast received
0 1 x x - - x -	Remote device MODBUS Send Event
0 1 x x x x x 1	Read Exception Sent (Code 1-3)
0 1 x x x x 1 x	Slave Abort Exception Sent (Code 4)
0 1 x x x 1 x x	Slave Busy Exception Sent (Code 5-6)
0 1 x x 1 x x x	Slave Program NAK Exception Sent (Code 7)
0 1 x 1 x x x x	Write Timeout Error Occurred
0 1 1 x x x x x	Currently in Listen Only Mode
0 0 0 0 0 1 0 0	Remote device entered Listen Only Mode
0 0 0 0 0 0 0 0	Remote device initiated communication restart

Table 4: *Event Bytes Coding*

- **Remote device Modbus Receive Event**
  The remote device stores this kind of event every time it received a request. The event is stored BEFORE the device starts the processing.

- **Remote device Modbus Sent Event**
  This event is stored by the remote device AFTER processing a request. This happens always independent of the response state (normal response, exception response or no response).

- **Remote device entered Listen Only Mode**
  The remote device records a switch into Listen Only Mode by storing code 04_h in the event queue.

- **Remote device initiated Communication Restart**
  The slave stores this event when its communication port was restarted e.g. by using Diagnostics function (08), sub-function 00 01 (Restart Communications Option). Depending on a data field in that Diagnostics request the event queue is cleared before the new event is stored.

Example

The following example shows a request to read the communication event log in remote device 19.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**13**	Address	**13**
Function	**0C**	Function	**0C**
CRC16 low	**0C**	Byte Count	**08**
CRC16 high	**85**	Status high	**00**
		Status low	**00**
		Event Count high	**01**
		Event Count low	**08**
		Message Count high	**01**
		Message Count low	**21**
		Event 0	**04**
		Event 1	**00**
		CRC16 low	**49**
		CRC16 high	**B9**

The remote device returns a status of 0000_h (device is not busy). The event count shows 0108_h or 264 counted events. And the message counter says that the device has processed 0121_h or 289 messages. According to the byte counter there are two additional events $(8 - 6 = 2)$.
The first or most recent communication event (Event 0) 04_h indicates the device has currently entered the Listen Only Mode[3]. The previous event is displayed as Event 1 and shows that the device has received a Communications Restart (00_h).

15 (0x0F) Write Multiple Coils

This function complements the Write Single Coil (which can set only one coil each time) and allows to force a sequence of consecutive coils to either ON or OFF. Coils are addressed starting at zero. Coil 1 therefore is referred as address 0.
The several ON/OFF states are packed as a byte sequence, starting with bit 0 in the first byte as state for the first coil and so on. A set bit (logical 1) forces the Coil (digital output) to be ON, logical 0 to be OFF.

Request

Function code	1 Byte	**0x0F**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0001 to 0x07B0
Byte Count	1 Byte	N*
Output values	N* 1 Byte	

[3]In the original MODBUS Application Protocol V1_1b3 this event is displayed as hex 20 which seems a typo!

The maximum quantity of 07B0$_h$ (decimal 1968) coils is specified by the maximum length of a Modbus RTU telegram which is 255 bytes.
Less the address, function number, start address, quantity, byte count and CRC16 checksum the remaining data size is 246 bytes or 1968 bits. N means the quantity of coils / 8. If:

$$\frac{N}{8} \neq 0 \Rightarrow N = N + 1$$

Response

Function code	1 Byte	**0x0F**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0000 to 0xFFFF

The remote device responses by returning the function code, start address and quantity of outputs.

Error

Function code	1 Byte	**0x8F**
Exception code	1 Byte	1, 2, 3, 4

Example

The following example shows how to set 10 coils in remote device 23 starting with coil number 40. The desired output should be:

Coil	40	41	42	43	44	45	46	47	48	49
State	ON	ON	OFF	ON	ON	OFF	OFF	OFF	ON	ON

The request needs 2 bytes to hold the 10 coil output bits.

	Byte 1								Byte 2							
Coil	47	46	45	44	43	42	41	40	-	-	-	-	-	-	49	48
Bit	0	0	0	1	1	0	1	1	0	0	0	0	0	0	1	1

The first byte addresses the coils 40 to 47 and is transmitted as 1B$_h$ whereas the least significant bit (LSB) holds the state of coil 40. The coils 48 and 49 are transmitted as 03$_h$ with the LSB addressing coil 48. Unused bits (the bits 2...7 in the second byte) should be zero filled.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**17**	Address	**17**
Function	**0F**	Function	**0F**
Starting Address high	**00**	Starting Address high	**00**
Starting Address low	**27**	Starting Address low	**27**
Output Quantity high	**00**	Output Quantity high	**00**
Output Quantity low	**0A**	Output Quantity low	**0A**
Byte Count	**02**	CRC16 low	**67**
Output Byte 1	**1B**	CRC16 high	**31**
Output Byte 2	**03**		
CRC16 low	**4F**		
CRC16 high	**7E**		

16 (0x10) Write Multiple Registers

Similar to Write Single Register but allowing to write a consecutive number of 1 to 123 registers (the maximum data payload for this Modbus telegram is 246 bytes) in one request. The data is passed as two bytes for every register in the request data field. The address counting starts from zero.

Request

Function code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2 x N*
Register Values	N* 2 Bytes	Value

N* is quantity of registers (not bytes).
Please note! Even that the number of registers is passed as register quantity the function still demands a byte count.

Response

Function code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B

The remote device responses by returning the function code, start address and quantity of registers.

Error

Function code	1 Byte	**0x90**
Exception code	1 Byte	1, 2, 3, 4

Here is an example to write 3 registers in the remote device 23. The registers start at address 50, the register values are 100 (0064$_h$), 200 ($00C8_h$) and 300 ($012C_h$).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**17**	Address	**17**
Function	**10**	Function	**10**
Starting Address high	**00**	Starting Address high	**00**
Starting Address low	**31**	Starting Address low	**31**
Register Quantity high	**00**	Register Quantity high	**00**
Register Quantity low	**03**	Register Quantity low	**03**
Byte Count	**06**	CRC16 low	**D3**
Register Value high	**00**	CRC16 high	**31**
Register Value low	**64**		
Register Value high	**00**		
Register Value low	**C8**		
Register Value high	**01**		
Register Value low	**2C**		
CRC16 low	**70**		
CRC16 high	**97**		

17 (0x11) Report Slave ID (Serial line only)

This function is intended to query the type, running status and other information of a given remote device. The response includes bytes which are specified by Modbus as device depending. Processing the data therefore is not standardized and difficult. It comes as no surprise that this function is rarely supported in the slaves and even rarer in the master devices.

Request

Function code	1 Byte	**0x11**

The data returned in a normal response depends on the device. A normal response looks like:

Response

Function code	1 Byte	**0x11**
Byte Count	1 Byte	
Remote Device ID	Device specific	
Running Status	1 Byte	0x00=OFF, 0xFF=ON
Additional Data		

Error

Function code	1 Byte	**0x91**
Exception code	1 Byte	1 or 4

The following example shows the response of a hypothetical device on address 100. The additional data field returns the vendor name, device model name and serial number as a comma separated ASCII string. For example:

```
IFTOOLS,IF-Sensor,101305023
```

The data in hex:

```
49 46 54 4f 4f 4c 53 2c
49 46 2d 53 65 6e 73 6f
72 2c 31 30 31 33 30 35
30 32 33
```

The byte count is 29 (Remote Device ID=1 , Running status=1 and data field length=27). The individual (hypothetical) device ID is 55_h. The device itself is running (Running State FF_h).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**64**	Address	**64**
Function	**11**	Function	**11**
CRC16 low	**EB**	Byte Count	**1D**
CRC16 high	**7C**	Remote Device ID	**55**
		Running Status	**FF**
		Additional Data	**see above...**
		CRC16 low	**58**
		CRC16 high	**FC**

Please note! This is just an example! A correct evaluation by the application is only possible if you know, what kind of data the requested remote device returns!

20 (0x14) Read File Record

This (and Write File Record too) is one of the more obscure commands in Modbus and rarely provided by Modbus devices. Nevertheless there exist devices which exclusively use the function to make some internal information accessible.
As the name suggests the function performs a file record read. A file is organized as a list of records (up to 10000 and indexed from 0) which are referenced via sub-request fields. The function allows to request one or more records (or data blocks) in one command by string several sub-request fields together. Every sub-request field is 7 bytes long and serves to identify the file, data location and data size within the file. It requires the following information:

The reference type	1 Byte	Always 6
The file number (see note below)	2 Bytes	0x0001 to 0xFFFF
The starting record number (address) within the file	2 Bytes	0...9999 (0x270F)
The record length to be read	2 Bytes	See note below!

Please note!

Even if the allowed file number range is up to 0xFFFF, [2] recommend not to use numbers greater than 10 to avoid interoperability with legacy equipment.
And: The number of requested bytes given by the information in the sub-request fields must not exceed the maximum allowable Modbus message length of 256 bytes!

Request

Function code	1 Byte	**0x14**
Byte Count	1 Byte	7 to 245 bytes
SubReq x, Ref. Type	1 Byte	always 6
SubReq x, File Number	2 Bytes	0x0001 to 0xFFFF
SubReq x, Record Number	2 Bytes	0x0000 to 0x270F
SubReq x, Record Length	2 Bytes	N
SubReq x+1, ...		

A normal response contains a series of data, one for each sub-request.

Response

Function code	1 Byte	**0x14**
Response Data Length	1 Byte	7 to 245 bytes
SubReq x, File resp. length	1 Byte	7 to 245 bytes
SubReq x, Reference Type	1 Byte	always 6
SubReq x, Record Data	N * 2 Bytes	
SubReq x+1, ...		

The Response Data Length field is the total number of bytes of all sub-responses. Every sub-response also contains an individual length field of its own (File resp. length).

Error

Function code	1 Byte	**0x94**
Exception code	1 Byte	1, 2, 3, 4, 8

In the following example the master requests two records from remote device 8. The first record is in file 4 and starts at address 1. It's size is 2 registers (or 2 x 16 byte). The second record is in file 3. It starts at address 9 and also contains 2 registers. Note that the reference is always 6.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**08**	Address	**08**
Function	**14**	Function	**14**
Byte Count	**0E**	Resp. Data Len	**0C**
SubReq1, Ref. Type	**06**	SubReq1, File resp. Len	**05**
SubReq1, File No. high	**00**	SubReq1, Ref. Type	**06**
SubReq1, File No. low	**04**	SubReq1, Reg. Data high	**0D**
SubReq1, Rec. No. high	**00**	SubReq1, Reg. Data low	**FE**
SubReq1, Rec. No. low	**01**	SubReq1, Reg. Data high	**00**
SubReq1, Rec. Len high	**00**	SubReq1, Reg. Data low	**20**
SubReq1, Rec. Len low	**02**	SubReq2, File resp. Len	**05**
SubReq2, Ref. Type	**06**	SubReq2, Ref. Type	**06**
SubReq2, File No. high	**00**	SubReq2, Reg. Data high	**33**
SubReq2, File No. low	**03**	SubReq2, Reg. Data low	**CD**
SubReq2, Rec. No. high	**00**	SubReq2, Reg. Data high	**00**
SubReq2, Rec. No. low	**09**	SubReq2, Reg. Data low	**40**
SubReq2, Rec. Len high	**00**	CRC16 low	**B0**
SubReq2, Rec. Len low	**02**	CRC16 high	**A7**
CRC16 low	**22**		
CRC16 high	**AF**		

As mentioned before: Function 20 is very rarely used. But it provides a universal method to make different kinds of data structures accessible by a single Modbus function[4]. For instance to map data located at different addresses (in a device memory) into one range. On the downside is: It's a complicated call and takes great effort to unravel the data in the response.

21 (0x15) Write File Record

Write File Function is the counterpart of Read File Record described in the preceding section. It writes the content of one or more consecutive registers (or data blocks) into an external (or extended memory) file. Like in the Read File Record function the data is organized as sub-requests. Every sub-request is followed by the data to be written.

The reference type	1 Byte	Always 6
The file number	2 Bytes	0x0001 to 0xFFFF
The starting record number (address) within the file	2 Bytes	0...9999 (0x270F)
The record length to be written	2 Bytes	N
The data to be written	N * 2 Bytes	

Please note! As usual the passed sub-requests must not exceed the maximum length of a Modbus telegram. And: As in Read File Record the file number must not exceed 10!

[4]The original name of this function was 'Read General Reference' which seems to describe it more precise.

Request

Function code	1 Byte	**0x15**
Request data length	1 Byte	9 to 251 bytes
SubReq x, Ref. Type	1 Byte	always 6
SubReq x, File Number	2 Bytes	0x0001 to 0xFFFF
SubReq x, Record Number	2 Bytes	0x0000 to 0x270F
SubReq x, Record Length	2 Bytes	**N**
SubReq x, Record data	**N** * 2 Bytes	
SubReq x+1, ...		

The normal response is the echo of the request.

Response

Function code	1 Byte	**0x15**
Request data length	1 Byte	9 to 251 bytes
SubReq x, Ref. Type	1 Byte	always 6
SubReq x, File Number	2 Bytes	0x0001 to 0xFFFF
SubReq x, Record Number	2 Bytes	0x0000 to 0x270F
SubReq x, Record Length	2 Bytes	**N**
SubReq x, Record data	**N** * 2 Bytes	
SubReq x+1, ...		

Error

Function code	1 Byte	**0x94**
Exception code	1 Byte	1, 2, 3, 4, 8

An example of a request to write one group of references into remote device 8 is listed below. The group consists of three registers in file 4, starting at register 7 (address 0007).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**08**	Address	**08**
Function	**15**	Function	**15**
Byte Count	**0D**	Byte Count	**0D**
SubReq1, Ref. Type	**06**	SubReq1, Ref. Type	**06**
SubReq1, File No. high	**00**	SubReq1, File No. high	**00**
SubReq1, File No. low	**04**	SubReq1, File No. low	**04**
SubReq1, Rec. No. high	**00**	SubReq1, Rec. No. high	**00**
SubReq1, Rec. No. low	**07**	SubReq1, Rec. No. low	**07**
SubReq1, Rec. Len high	**00**	SubReq1, Rec. Len high	**00**
SubReq1, Rec. Len low	**03**	SubReq1, Rec. Len low	**03**
SubReq1, Reg. Data high	**06**	SubReq1, Reg. Data high	**06**
SubReq1, Reg. Data high	**AF**	SubReq1, Reg. Data high	**AF**
SubReq1, Reg. Data high	**04**	SubReq1, Reg. Data high	**04**
SubReq1, Reg. Data high	**BE**	SubReq1, Reg. Data high	**BE**
SubReq1, Reg. Data high	**10**	SubReq1, Reg. Data high	**10**
SubReq1, Reg. Data high	**0D**	SubReq1, Reg. Data high	**0D**
CRC16 low	**10**	CRC16 low	**10**
CRC16 high	**5D**	CRC16 high	**5D**

Continued on next page

Continued from previous page

22 (0x16) Mask Write Register

This function can be used to set or clear certain bits in a holding register by a logical operation of the content with an AND/OR mask register.
To clear individual bits in the holding register is achieved by clear these bits in the AND mask. To set bits by set the desired bits in the OR mask.
Registers are addressed starting at zero, register 1 therefore means register address 0.

Request

Function code	1 Byte	**0x16**
Reference Address	2 Bytes	0x0000 to 0xFFFF
AND Mask	2 Bytes	0x0000 to 0xFFFF
OR Mask	2 Bytes	0x0000 to 0xFFFF

The normal response is the echo of the request.

Response

Function code	1 Byte	**0x16**
Reference Address	2 Bytes	0x0000 to 0xFFFF
AND Mask	2 Bytes	0x0000 to 0xFFFF
OR Mask	2 Bytes	0x0000 to 0xFFFF

Error

Function code	1 Byte	**0x96**
Exception code	1 Byte	1, 2, 3, 4

The following example sets the first 4 bits and clears the upper 2 bits in register 5 of the remote device 20. The bits 5...14 remain in their current state.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**14**	Address	**14**
Function	**16**	Function	**16**
Reference address high	**00**	Reference address high	**00**
Reference address low	**04**	Reference address low	**04**
AND mask high	**3F**	AND mask high	**3F**
AND mask low	**FF**	AND mask low	**FF**
OR mask high	**00**	OR mask high	**00**
OR mask low	**0F**	OR mask low	**0F**
CRC16 low	**BA**	CRC16 low	**BA**
CRC16 high	**D5**	CRC16 high	**D5**

Let us assume the current register content is 0x7CA9.

```
0x7CA9 = 0111 1100 1010 1001
AND    = 0011 1111 1111 1111
      => 0011 1100 1010 1001
OR     = 0000 0000 0000 1111
      => 0011 1100 1010 1111
```

The result after apply the command is 0x3CAF.

23 (0x17) Read/Write Multiple Registers

This function combines a read and write operation whereas the write is executed (in the remote device) before the read. The registers for the read and write access (mostly holding registers) are addressed from zero. A register start address of 15 therefore is register number 16.

Request

Function code	1 Byte	**0x17**
Read Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity to Read	2 Bytes	0x0001 to 0x007D
Write Starting Address	2 Bytes	0x0001 to 0xFFFF
Quantity to Write	2 Bytes	0x0001 to 0x0079
Write Byte Count	1 Byte	2 x N
Write Register Values	N * 2 Bytes	

In a normal response the remote device returns the data that were read.

Response

Function code	1 Byte	**0x17**
Byte Count	1 Byte	2 x N
Read Register Values	N * 2 Bytes	

Error

Function code	1 Byte	**0x97**
Exception code	1 Byte	1, 2, 3, 4

The following example reads 2 registers starting at register 100 and writes 1 register (value 0xABCD) starting at register 200. The remote device address is 5.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**05**	Address	**05**
Function	**17**	Function	**17**
Read Start Address high	**00**	Byte Count	**04**
Read Start Address low	**63**	Read Register high	**00**
Read Quantity high	**00**	Read Register low	**01**
Read Quantity low	**02**	Read Register high	**00**
Write Start Address high	**00**	Read Register low	**02**
Write Start Address low	**C7**	CRC16 low	**6C**
Write Quantity high	**00**	CRC16 high	**E6**
Write Quantity low	**01**		
Write Byte Count	**02**		
Write Register high	**AB**		
Write Register low	**CD**		
CRC16 low	**47**		
CRC16 high	**9C**		

24 (0x18) Read FIFO Queue

This function reads the contents of a FIFO (First-In-First-Out) register queue in a remote device. The FIFO is addressed like a holding register but returns number of the available FIFO (register) items followed by up to 31 register values.
Please note! According to the Modbus specifications the function reads the queue content WITHOUT clearing them[5].

Request

Function code	1 Byte	**0x18**
FIFO Pointer Address	2 Bytes	0x0000 to 0xFFFF

In the request we do not have to pass the number of registers we want to read. Just the register address holding the FIFO queue is enough.

Response

Function code	1 Byte	**0x18**
Byte Count	2 Bytes	0x0000 to 0xFFFF
FIFO Count	2 Bytes	`0 < N < 31`
FIFO Value Registers	N * 2 Bytes	

N means the available FIFO values. In a normal response N is always ≤ 31. If the FIFO holds more than 32 items (N is > 32) an exception response is returned with error code 3 (Illegal Data Value)[6].

[5]Question: How is the read data removed from the queue? Does this happen automatically by the read access?
[6]How can the application deal with such a situation?

Error

Function code	1 Byte	**0x98**
Exception code	1 Byte	1, 2, 3, 4

In the example below we access the FIFO queue in remote device 5. The FIFO is accessible via register address 0x04DE. The FIFO holds 2 register values. The first register contents 0x01B8 (decimal 440), the second one 0x1204 (decimal 4612).

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**05**	Address	**05**
Function	**18**	Function	**18**
FIFO Pointer Address high	**04**	Byte Count high	**00**
FIFO Pointer Address low	**DE**	Byte Count low	**06**
CRC16 low	**02**	FIFO Count high	**00**
CRC16 high	**77**	FIFO Count low	**02**
		FIFO Reg1 high	**01**
		FIFO Reg1 low	**B8**
		FIFO Reg2 high	**12**
		FIFO Reg2 low	**04**
		CRC16 low	**59**
		CRC16 high	**6D**

43 (0x2B) Encapsulated Interface Transport

This function was mainly developed for tunneling other protocols inside Modbus telegrams, especially CanOpen. Additional it provides a feature to query device information like model number, serial number, etc. The kind of the encapsulated data is specified by the so called MEI (Modbus Encapsulated Interface) type field which follows immediately after the Modbus function number. It is a unique number assigned by Modbus with a reserved range from 0 to 255 except for the MEI type number 13 and 14 which have an already predefined meaning[7].
A simple request without digging into the MEI data looks like:

Request

Function code	1 Byte	**0x2B**
MEI Type	1 Byte	0x0D or 0x0E
MEI type specific data	n Bytes	

[7]For example the MEI Type 13 (0x0D) is a MODBUS Assigned Number licensed to CiA for the CANopen General Reference. MEI Type 14 serves as a general function to provide detail device information.

Response

Function code	1 Byte	**0x2B**
MEI Type	1 Byte	Echo of MEI type in request
MEI type specific data	n Bytes	

Error

Function code	1 Byte	**0xAB**
Exception code	1 Byte	1, 2, 3, 4

The MEI transport service is meant to be interface independent. Thus any specific behaviour or policy required by the interface must handled by the interface itself. That includes the transaction processing, error handling and so on.

MEI 13 - CANopen Request and Response PDU

The MEI type specific data is defined by CANopen and not part of the Modbus specification. We therefore can only provide some small examples to give a first impression how MEI works here. For more information see section Further links. A CANopen related request looks like:

Function code	1 Byte	**0x2B**
MEI Type	1 Byte	**0x0D**
CANopen Protocol option fields	2 to 5 Bytes	
CANopen Address and data fields	N Bytes	

The content of the CANopen option and data/address fields determine how the interface has to handle the message and cannot be part of this article. The following telegram examples are intended only to illustrate the tunneling from the Modbus point of view.

The telegram below reads out the CANopen object 6042_h:00_h. The slave (Modbus) address is 5. They are drawn from de.nanotec.com.

SA	FC	MEI	CANopen Data										CRC	
05	2B	0D	00	00	01	60	42	00	00	00	00	02	7F	0F

The response looks like:

SA	FC	MEI	CANopen Data												CRC	
05	2B	0D	00	00	01	60	42	00	00	00	00	02	00	00	60	34

MEI 14 - Read Device Identification

This function is used to query the identification and other information according to the physical and function description of a remote device. It does not provide a running state as function Report Slave ID.

The Read Device Identification function divides the searchable information in a number of data elements called objects. Each object has it own ID and belongs to one of four categories.

Object ID	Object Name Description	Type	MO	Category
0x00	VendorName	ASCII String	Mandatory	Basic
0x01	ProductCode	ASCII String	Mandatory	Basic
0x02	MajorMinorRevision	ASCII String	Mandatory	Basic
0x03	VendorURL	ASCII String	Optional	Regular
0x04	ProductName	ASCII String	Optional	Regular
0x05	ModelName	ASCII String	Optional	Regular
0x06	UserApplicationName	ASCII String	Optional	Regular
0x07 .. 0x7F	Reserved		Optional	Regular
0x80 .. 0xFF	Private objects may be optional defined. The range 0x80...0xFF is product dependant.	Device dependent	Optional	Extended

There are three categories displayed in the table above. A fourth one allows the individual access of a specific single object (category independent) by passing the object ID in the request. The categories are numbered from 1 to 4.

1 Get the **basic** device information

2 Get the **regular** device information

3 Get the **extended** device information

4 Get one **specific** device information

Request

Function code	1 Byte	**0x2B**
MEI Type	1 Byte	0x0E
Read Device ID Code	1 Byte	01/02/03/04
Object ID	1 Byte	0x00 to 0xFF

The Object ID field has a double meaning. When requesting a specific device information it holds the Object ID of the required data. For example 02_h for the Major Minor Revision.
If the master requests a category the response may not fit in one telegram and several transactions may are necessary, see examples below. The master therefore set the Object ID to 0 in the very first request.

In case of further transactions the master receives the next object ID to be requested from the slave in the response field `Next Object ID` additional with a FF_h in the `More follows` field (see response table below). The master then continues the category request with the new object ID.

Response

Function code	1 Byte	**0x2B**
MEI Type	1 Byte	0x0E
Read Device ID Code	1 Byte	01/02/03/04
Conformity Level	1 Byte	0x01 or 0x02 or 0x03 or 0x81 or 0x82 or 0x83
More follows	1 Byte	0x00 or 0xFF
Next Object ID	1 Byte	Object ID number
Number of Objects	1 Byte	
List of Objects ...		
Object ID	1 Byte	
Object Length	1 Byte	
Object Value	x Bytes	Depending on object

The `Conformity Level` field normally contains the value of the requested category. But if the master requests a category which is not supported by the remote device, the slave responses in accordance with its actual conformity level. Since the answer is not as expected, the `Confirmation Level` field tells the master how to interpret the data instead.
A response can contain one or more information objects. While objects have no predefined length (basic and regular objects are ASCII strings), every object in the response has also a length descriptor.

Error

Function code	1 Byte	**0xAB**
Exception code	1 Byte	1, 2, 3, 4

In case of an illegal category the remote device responses with an exception code 03 (illegal data value). If the master requests a special device information which is not supported by the slave, the response is an exception code 02 (illegal data address).

Examples

In the following example the requested information (basic device information, category 1) is sent in one response.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**08**	Address	**08**
Function	**14**	Function	**14**
Read Device ID Code	**01**	Read Device ID Code	**01**
Object ID	**00**	Conformity Level	**01**
CRC16 low	**1D**	More follows	**00**
CRC16 high	**88**	Next Object ID	**00**
		Number of Objects	**03**
		Object1 ID	**00**
		Object1 Lenght	**07**
		Object1 Value	**"IFTOOLS"**
		Object1 ID	**01**
		Object1 Lenght	**06**
		Object1 Value	**"SENSOR"**
		Object1 ID	**02**
		Object1 Lenght	**05**
		Object1 Value	**"V1.01"**
		CRC16 low	**A6**
		CRC16 high	**A7**

The next examples shows again the request of the basic device information (category 1) but now assumes that the information does not fit in one Modbus frame and therefore needs two transactions.
The first transaction only contains the information for the objects 0 (Vendor Name) and 1 (Product Code). It looks like:

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**08**	Address	**08**
Function	**14**	Function	**14**
Read Device ID Code	**01**	Read Device ID Code	**01**
Object ID	**00**	Conformity Level	**01**
CRC16 low	**1D**	More follows	**FF**
CRC16 high	**88**	Next Object ID	**02**
		Number of Objects	**03**
		Object1 ID	**00**
		Object1 Lenght	**19**
		Object1 Value	**"The␣ Fieldbus␣ Company␣ Ltd."**
		Object1 ID	**01**
		Object1 Lenght	**16**
		Object1 Value	**"Fieldbus␣ Company␣ Typ␣1"**
		CRC16 low	**66**
		CRC16 high	**DE**

The field `More follows` indicates now, that there are pending information starting with the object ID 02 (field `Next Object ID`) and the master therefore initiates a second request.

Request		Response	
Field	**Hex**	**Field**	**Hex**
Address	**08**	Address	**08**
Function	**14**	Function	**14**
Read Device ID Code	**01**	Read Device ID Code	**01**
Object ID	**02**	Conformity Level	**01**
CRC16 low	**1D**	More follows	**00**
CRC16 high	**88**	Next Object ID	**00**
		Number of Objects	**03**
		Object1 ID	**02**
		Object1 Lenght	**05**
		Object1 Value	**"V1.01"**
		CRC16 low	**02**
		CRC16 high	**38**

Since this is the last transaction (no further information), the `More follows` field is again 00. The same is valid for the `Next Object ID` which is also set to 00 by the responding device.

Modbus Exception Codes

When the master sends a request to a slave, four possible events may occur.

- **Normal response**
  The slave receives the master request without an error and can handle it properly. It then returns a normal response.

- **No response (timeout)**
  The slave with the given address does not exist or detects an error in the request. In both cases no response is returned to the master.

- **Erroneous response**
  The master detects a faulty response (invalid telegram frame, checksum error). The reaction depends on the application, usually the master repeats the request several times.

- **Exception response**
  The slave receives the request error-free but cannot process the request properly. For instance if the master tries to read/write a not existing IO address. The slave then will return an exception response to inform the master about the error.

All Modbus function codes in a request or response have a most significant bit (MSB) of 0 (the function code is below hex 80). In case of an error, the slave set the MSB to 1, so the function code is greater as hex 80. The data field contains the error code as a single byte.

Address	Function Code	Data	Checksum
Echo address	Echo code + 80hex	Exception code	Checksum

The following table lists all exception codes. Please note that not all codes are supported by every application.

Code	Name	Description
01	ILLEGAL FUNCTION	The slave does not support the requested function or is not in the state to handle it.
02	ILLEGAL DATA ADDRESS	The requested data address is not in the allowed IO range of the slave or not writable (read-only).
03	ILLEGAL DATA VALUE	Invalid quantity of requested data.
04	SLAVE DEVICE FAILURE	An unrecoverable error occurs while the slave performs the requested action.
05	ACKNOWLEDGE	Private function code (application dependent). A slave can use this code to signal the master that it needs more time to process the request. The master itself then can issue one or more Poll Program Complete message(s) (also private) to determine if the processing is complete.
06	SLAVE DEVICE BUSY	The slave is busy due to processing a long duration program command. This exception response informs the master to repeat the request later again.
08	MEMORY PARITY ERROR	Used in conjunction with functions 20 and 21 and reference type 6. Indicates a consistency check failure in the extended file area.
0A	GATEWAY PATH UNAVAILABLE	Gateway cannot route the message. This usually means a misconfigured or overloaded gateway.
0B	GATEWAY TARGET DEVICE FAILED TO RESPONSE	Error response from a gateway that the addressed device does not response.

Table 5: *Modbus Error Codes*

Distinguish requests from responses

The Modbus specification does not provide a telegram flag or information field to determine the sender (master or slave) of the telegram.
An external observer (for instance an field-bus analyzer tapping a 2-wire half-duplex RS485 bus) can only figure out the origin of a data sequence by examine the telegram content, telegram flow and sometimes only by the timing.
The following table compares the length of both, request and response. The displayed lengths include the address and checksum bytes!

Function	Request	Response
01 Read Coils	8	6...255
02 Read Discrete Input	8	6...255
03 Read Holding Registers	8	7,9...255
04 Read Input Registers	8	7,9...255
05 Write Single Coil	8	8
06 Write Single Register	8	8
07 Read Exception Status	4	5
08 Diagnostics	8,10...256	8,10...256
11 Get Comm Event Counter	4	8
12 Get Comm Event Log	4	12...76
15 Write Multiple Coils	11...255	8
16 Write Multiple Registers	11,13...255	8
17 Report Slave ID	4	7...256
20 Read File Record	12,19...250	9,11...255
21 Write File Record	14,16...254	14,16...254
22 Mask Write Register	10	10
23 Read/Write Multiple Registers	13,15...255	5,7...255
24 Read Fifo Queue	6	10,12...70
43 Encapsulated Interface Transport	6...256	6...256

Table 6: *Request and response telegram lengths*

Telegrams which are clearly distinguishable from each other (different length or size) are marked as dark gray. Other functions we can keep apart because the request has always an odd number of bytes whereas the response has always an even length. These are marked as light gray.
The remaining functions which we cannot lightly separate in requests or responses are remaining in white. We may keep apart some of them by looking for special data pattern (preset field contents) in the telegram sequence. But other functions even have identical requests and responses (the normal response is just the echo of the request), for example the basic functions 5 (Write Singe Coil) and 6 (Write Single Register).
We can interprete two such successive telegrams as a request and response - or - as two requests without response!

An observer can try to determine the origin of these telegrams by analysing the timing. If the time interval between the former and current telegram is between the specified response time, it can take it as a response, otherwise as a perhaps second try of the master.
Nevertheless there is still a chance, that the response just exceed the allowed response time.
And most important! All these described methods depend on the correctness of the transmitted telegram frames!

But what are the possibilities to assign single data bytes to the master or to a slave in a half-duplex bus where every bus participant shares the same two lines?

The IFTOOLS MSB-RS485-PLUS field-bus analyzer was specially developed to target this problem. The analyzer provides a so called segment mode analysis. This special feature allows to insert the analyzer into the bus between the master and the slaves. Working as a transparent data router between two bus segments (master segment and slave segment) the analyzer records not only all transmitted bus data with precise time information. It also detects the direction (or origin) of every data byte.
Thus even invalid or erroneous data sequences are assigned absolutely reliable to the master or slave segment - a crucial requirement when analyzing faulty Modbus ASCII and RTU transmissions.

Conclusion

To rely only on the analysis of the telegram content, telegram flow or timing may produce good in results in most cases but it fails if the bus transmission contains corrupted telegrams (by faulty bus participants or caused by neglected timing specifications). In this case a classification between sender and recipient cannot be made any longer since the initial data are already wrong.
Without any additional direction information (like as the segment mode provided by the IFTOOLS MSB-RS485 analyzers) the cause of the failure (master or slave) question remains open.

Further links

https://www.modbus.org
Modbus_over_serial_line_V1.pdf
Modbus_Application_Protocol_V1_1b3.pdf
Modbus on Wikipedia
IFTOOLS MSB-RS485 Analyzer
Modbus Master simulation via serial port

References

[1] MODBUS over Serial Line, Specification and Implementation Guide, V1.02

[2] Modbus Application Protocol Specification V1.1b3